



励志照亮人生 编程改变命运

# 零基础学 SQL



孙浏毅 等编著



## 20.5小时全程多媒体视频讲解

- ◎ 循序渐进：从数据库与数据表的创建开始讲解，逐步过渡到SQL语言的学习
- ◎ 内容全面：涵盖SQL语言数据查询、数据更新、数据控制等方面的内容
- ◎ 便于学习：对SQL语句采用语法规则、语法说明、实例代码、实例讲解、显示结果的结构阐述，方便学习和查询
- ◎ 对比讲解：对比了SQL Server、Oracle和MySQL三种数据库在SQL实现上的差异
- ◎ 实例丰富：讲解每一个SQL语句时都提供了多个示例，全书贯穿示例达400余个
- ◎ 视频教学：配有20.5小时多媒体视频进行讲解，学习效果好



机械工业出版社  
China Machine Press



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

励志照亮人生 编程改变命运

# 零基础学 SQL

## 本书涵盖内容

- ◎ 概念模型、关系数据模型、关系模式
- ◎ SQL语言的分类和特点、SQL语言的书写规范、常用数据类型
- ◎ 数据库的创建和删除、数据表的创建和更新
- ◎ 数据记录、属性、字段、列、行、主键、外键、约束、索引
- ◎ 使用约束、使用索引
- ◎ 修改数据库中的表、删除数据库中的表
- ◎ 基本查询操作、比较查询、逻辑查询、空值查询、模糊查询
- ◎ 表中数据的排序与分组
- ◎ 多表连接查询与集合查询、相关子查询与多重子查询
- ◎ 常用函数的使用、视图的创建与维护
- ◎ 数据记录的增加、删除和修改
- ◎ 权限的授予与回收、事务的控制与管理
- ◎ PL/SQL的编写规范、块结构、基本要素、数据类型和控制结构
- ◎ 使用游标、异常处理
- ◎ 存储过程的创建与维护、函数的创建与维护、包的创建与维护
- ◎ 触发器创建与管理
- ◎ SQL语句性能优化、动态SQL、数据库的存取访问

客服热线：(010) 88378991, 88361066  
购书热线：(010) 68326294, 88379649, 68995259  
投稿热线：(010) 88379604  
读者信箱：hzsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书：[www.china-pub.com](http://www.china-pub.com)

封面设计：1-12 11.5



DVD-ROM



上架指导：计算机/数据库

ISBN 978-7-111-29198-5



9 787111 291985

定价：59.80 元（附光盘）



## 出版说明

“零基础学编程”系列图书自2008年上市以来就受到了广大读者的青睐。本系列中的很多图书一上市就登上了编程类图书销售排行榜的前列。目前本系列中的很多图书已经多次印刷。之所以有如此好的市场表现，是与本系列书的品质和在读者中较好的口碑分不开的。本系列书上市后受到了广大读者的好评，很多学校也将其作为教材使用。

为了使本系列书能紧跟技术趋势，更加适合读者学习和学校教学，我们结合最新技术和读者的建议，对本系列图书中的一些图书进行了改版（即第2版）。另外，还增加了一些新的品种，也一并放入本系列，以使本系列图书更加完善。

### 第2版图书所做的改进

第2版图书在第1版图书的基础上主要有以下改进：

- ☐ 增加了数小时的多媒体教学视频，使得学习更加直观和高效；
- ☐ 增加了课后习题，使得本系列书更加适合读者自我检测和学校教学使用；
- ☐ 专门制作了教学PPT，以便于相关专业老师教学使用；
- ☐ 增加了更多的项目实践内容，以增强实用性，提高读者的动手能力；
- ☐ 对图书的编排体例进行了梳理，以增强条理性和可读性；
- ☐ 对第1版图书的内容和结构有所调整，使得其更加合理和科学；
- ☐ 补充完善了一些新的内容，使其内容更加完善；
- ☐ 更正了第1版图书中出现的一些疏漏。

### 包括的书籍

本系列书本次推出11个品种。其中，标注了第2版的为第1版的改版，未标注的为本次新增加品种。具体如下：

《零基础学Java 第2版》

《零基础学Java Web开发 第2版》

《零基础学Visual C++ 第2版》

《零基础学Visual Basic 第2版》

《零基础学JavaScript 第2版》

《零基础学SQL Server 2008》

《零基础学SQL》

《零基础学Linux C程序设计》

《零基础学数据结构》

《零基础学算法》

《零基础学计算机英语》

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

## 零基础学SQL

### 光盘内容

本系列书有配套光盘。光盘的主要内容如下：

- ☐ 书中所涉及的程序源代码；
- ☐ 多媒体语音教学视频；
- ☐ 教学PPT；
- ☐ 免费赠送的大量编程电子图书和编程视频。

### 技术支持

本系列书有专门的技术论坛（<http://www.rzchina.net>）和QQ群（群号：21948169）。读者学习过程中若有疑问，可以在论坛上或者QQ群里提问交流。另外，论坛上还有一些技术教程、视频动画，读者可以免费下载使用。





## 前 言

在实际的应用开发中，无论是应用软件的开发人员还是数据库的管理人员都需要与数据库打交道，而SQL语言作为数据库语言之一，是关系数据库系统中最常用的一种语言。因此全面了解并掌握SQL语言无论是对于软件开发人员还是对数据库的管理人员都是非常重要的。

大多数有关数据库的图书一般都会涉及关系数据库基本理论、数据库的设计与管理等方面的内容，对SQL语句部分并没有详细的阐述。本书是一本从基础知识开始全面讲解SQL的图书，从最简单的数据查询开始，到对数据的排序分组，再到一些复杂的数据查询，例如，数据表的连接、子查询以及一些数据库中的常用函数，然后再渐进到PL/SQL编程基础，PL/SQL中常量、变量、记录、集合的声明和使用，以及控制结构、存储过程、函数、包、触发器的创建和使用，在SQL应用中介绍了查询优化、动态以及使用高级程序设计语言通过SQL实现对数据库中数据的存取访问，循序渐进、系统全面地介绍了SQL的相关知识。

### 本书特点

#### 1. 由浅入深，循序渐进

为了方便读者学习，本书从关系数据库的基础知识讲起，以易于安装的开源数据库MySQL 5.0为基础，从SQL语句的基本语法入手，从简单的数据检索到对数据的排序分组再到数据表的连接、子查询，然后再渐进到PL/SQL编程、查询优化、SQL应用开发。

#### 2. 内容全面，结构清晰

不同的数据库系统对SQL的支持和扩展存在很大差异。为了方便读者学习，本书充分对比SQL Server、Oracle、MySQL在SQL实现上的差异，进行详细讲解。读者通过本书，可以全面掌握三种数据库的SQL使用。同时，本书按照数据定义语言（DDL）、数据查询语言（DQL）、数据操作语言（DML）、数据控制语言（DCL）、事务控制语言、SQL的扩展PL/SQL编程的顺序对SQL语句进行全面的讲解。

#### 3. 实例丰富，结合实际

本书对于每一个SQL语句的讲解都配有多个实例，使读者在了解SQL语法的同时，能够真正掌握其用法。同时结合目前主流的数据库Microsoft SQL Server和Oracle，对SQL语句在这些数据库中的使用差别也都在实例中给出了相应的实现方法。

#### 4. 查询方便，便于自学

对于大部分的SQL语句采用“语法规则、语法说明、实例代码、实例讲解、显示结果”的结构进行阐述。对于每一个SQL和PL/SQL语句都进行了详细讲解，便于读者理解。在本书的附录中还提供了常用SQL语句和在MySQL、Microsoft SQL Server以及Oracle数据库中常用函数的对照表，

便于读者对每一部分进行查询和学习。

## 5. 内容充实，要点突出

本书系统全面地介绍了SQL的知识，对于需要注意和需要强调的内容，以“注意”、“说明”、“提示”三种形式给出。其中，“注意”表示在使用SQL语句中可能会出现以及在实际使用中需要避免的问题；“说明”表示文中讨论的一些比较重要的信息；“提示”给出一种建议和方法。

## 本书内容

第1章：主要介绍关系数据库中涉及的几个重要的概念，包括概念模型、关系数据模型和关系模式，并介绍了几种常用的关系数据库，最后以MySQL 5.0数据库为例，介绍了MySQL 5.0数据库的安装和使用方法。

第2章：对SQL语言进行介绍，讲述SQL语句的分类、特点以及SQL语言中常用数据类型，最后介绍了SQL语句书写规范。

第3章：主要介绍使用SQL语句以及在MySQL 5.0 Command Line Client窗口和MySQL 5.0的用户图形界面下创建和删除数据库的方法。

第4章：全面地介绍了数据表中涉及的数据类型和关系数据库中几个重要的概念，包括主键、外键、索引和约束及其使用方法。另外，还将介绍数据表的创建方法，以及如何对表中的记录进行增加和修改操作，如何删除数据表等内容。

第5章：主要介绍SELECT语句查询的基本操作，包括如何查询全部列的记录，查询表中指定的列，查询表中不重复的记录，使用列别名查询，对查询的记录进行算术运算以及数据库中连接多个字段的方法。

第6章：主要介绍如何使用WHERE子句查询表中满足条件的记录，包括比较查询、逻辑查询、空值查询和模糊查询。

第7章：介绍SQL语句的聚合函数和GROUP BY子句的使用以及如何使用ROLLUP关键字进行数据统计。最后还将介绍MySQL数据库、Oracle数据库以及Microsoft SQL Server数据库中限制结果集行数的方法。

第8章：主要介绍内连接、交叉连接、自连接以及外连接四种连接查询的使用方法以及并操作、交操作和差操作三种集合查询。

第9章：主要介绍SQL语句中的子查询。包括单行子查询、多行子查询、多列子查询、相关子查询、多重子查询以及在CREATE TABLE语句中使用子查询实现数据表的复制的方法。

第10章：主要以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为基础，讲解在这三种数据库中一些常用函数的功能及其使用方法。

第11章：介绍视图的作用以及如何创建和删除一个视图。

第12章：主要讲解如何使用INSERT INTO语句向数据表和视图中插入数据。

第13章：主要讲解如何使用UPDATE SET语句在数据表和视图中修改数据。

第14章：主要讲解如何使用DELETE语句在数据表和视图中删除数据。

第15章：主要介绍使用GRANT语句授予权限和使用REVOKE语句回收权限的方法，并以MySQL 5.0数据库为例，介绍在MySQL 5.0数据库中使用Administrator管理系统授予用户权限的方法。



第16章：介绍事务的概念、事务管理以及事务的并发控制。

第17章：讲解PL/SQL的概念、使用PL/SQL的原因、PL/SQL的编写规范以及常用的PL/SQL开发工具的简单介绍。

第18章：介绍PL/SQL的块结构和PL/SQL的基本要素，重点介绍过程性的语句中涉及的变量声明、数据类型、变量作用域以及数据类型相互转换等内容。

第19章：介绍PL/SQL中的控制结构的使用方法，主要包括分支结构、循环结构和顺序结构。

第20章：主要介绍如何使用显式游标进行多行数据的查询、游标FOR循环以及游标变量的使用，另外还将介绍游标属性以及嵌套游标的使用等内容。

第21章：介绍PL/SQL中有关异常处理的相关内容，包括使用异常处理的原因、如何声明、抛出、捕获和处理异常以及异常的处理机制，最后介绍使用异常的一些基本原则。

第22章：主要介绍存储过程的创建、调用、参数模式以及参数传递方面的内容。

第23章：主要介绍函数的创建、调用、参数传递以及在SQL语句中如何调用函数等内容。最后，还将对存储过程和函数之间的异同进行了比较。

第24章：主要介绍包的创建、包中公有元素的调用、包中子程序的重载以及包的删除方法。最后还会介绍Oracle数据库中常用内置系统包。

第25章：主要介绍触发器的类型及其用途、触发器的创建和维护。

第26章：介绍一些有关SQL语句查询优化的方法。

第27章：介绍实现动态SQL语句的方法。

第28章：主要介绍数据库的存取访问方法。本章从数据库应用系统结构入手，介绍数据库应用系统结构的4种基本结构，然后介绍几种常用的数据库连接访问技术，最后通过一种高级程序设计语言Java与一个数据库MySQL 5.0的连接和开发的例子介绍如何使用程序设计语言实现对数据库的连接和访问。

附录中为常用SQL语句和常用函数对照。

## 本书适合的读者

- ☐ SQL语言的初学者
- ☐ 大中专院校计算机专业的学生和教师
- ☐ 数据库管理和开发人员
- ☐ 程序设计和软件开发人员
- ☐ 社会培训学生

## 本书作者

本书由孙浏毅主笔编写，同时参与编写和资料整理的有刘亮亮、丁士锋、何涛发、陈杰、黄曦、罗嘉、段春江、韩红宇、李嵩峰、莫光胜、王天国、李蓉、吴荣、宋祥亮、刘宇、吕晓鹏、王大伟、吴小平、张卫忠、施佳鹏、王嘉、吴雪、阳婷、张秀妍、王江、王志永、杨红、郑维龙、王松、张文。

编者

## 目 录

### 出版说明 前言

## 第一篇 关系数据库与SQL语言

第1章 关系数据库介绍	1
1.1 数据模型	1
1.1.1 概念模型	1
1.1.2 关系数据模型	3
1.2 关系模式	3
1.3 常用关系数据库	4
1.3.1 Oracle数据库	4
1.3.2 Microsoft SQL Server数据库	4
1.3.3 MySQL数据库	4
1.3.4 PostgreSQL数据库	5
1.4 安装与使用MySQL 5.0数据库	5
1.4.1 安装MySQL 5.0	5
1.4.2 安装用户图形界面	7
1.4.3 运行MySQL 5.0	8
1.5 小结	9
第2章 SQL语言概述	10
2.1 SQL语言介绍	10
2.2 SQL语句的分类	11
2.3 SQL语言的特点	11
2.4 常用数据类型	12
2.4.1 整数类型与浮点类型	12
2.4.2 数值类型	13
2.4.3 字符类型	13
2.4.4 日期与时间类型	14
2.4.5 二进制类型	15



2.5 SQL语句书写规范 .....	15
2.6 小结 .....	16

## 第二篇 数据库与数据表的创建和管理

第3章 数据库的创建与删除 .....	17
3.1 创建数据库 .....	17
3.1.1 使用SQL语句创建数据库 .....	17
3.1.2 在MySQL 5.0 Command Line Client窗口下创建数据库 .....	18
3.1.3 在MySQL 5.0用户图形界面中创建数据库 .....	18
3.2 删除数据库 .....	19
3.2.1 使用SQL语句删除数据库 .....	19
3.2.2 在MySQL 5.0 Command Line Client窗口下删除数据库 .....	19
3.2.3 在MySQL 5.0用户图形界面中删除数据库 .....	20
3.3 小结 .....	21
第4章 数据表的创建与更新 .....	22
4.1 数据库中的表 .....	22
4.1.1 数据记录、属性、字段、列和行 .....	22
4.1.2 主键 .....	23
4.1.3 外键 .....	23
4.1.4 索引 .....	24
4.1.5 约束 .....	24
4.2 创建数据表 .....	25
4.3 使用约束 .....	27
4.3.1 唯一约束 .....	27
4.3.2 主键约束 .....	28
4.3.3 外键约束 .....	30
4.3.4 检查约束 .....	31
4.3.5 非空约束 .....	32
4.4 使用索引 .....	33
4.4.1 索引的分类 .....	33
4.4.2 创建与删除索引 .....	33
4.5 修改数据库中的表 .....	35
4.5.1 向表中增加一列 .....	35
4.5.2 增加一个约束条件 .....	36
4.5.3 增加一个索引 .....	37
4.5.4 修改表中的某一列 .....	38
4.5.5 删除表中某一列 .....	38
4.5.6 删除一个约束条件 .....	39
4.6 删除数据库中的表 .....	39

4.7 数据库test_STInfo中的表	39
4.7.1 学生信息表T_student	40
4.7.2 课程信息表T_curriculum	40
4.7.3 成绩信息表T_result	40
4.7.4 教师信息表T_teacher	41
4.7.5 院系信息表T_dept	41
4.7.6 计算机系教师信息表T_CSteacher	41
4.8 小结	42

### 第三篇 数据查询

第5章 基本查询操作	43
5.1 查询全部列的记录	43
5.2 查询表中指定的列	44
5.3 查询表中不重复的记录	45
5.4 使用列别名查询	46
5.5 对查询的记录进行算术运算	47
5.6 使用连接符 (  ) 连接字段	47
5.7 关于NULL值	48
5.8 小结	49
第6章 使用WHERE子句查询表中满足条件的记录	50
6.1 比较查询	50
6.1.1 算术比较运算符	50
6.1.2 BETWEEN...AND运算符查询指定条件范围的记录	51
6.1.3 IN运算符查询与列表匹配的记录	52
6.1.4 字符串比较	53
6.1.5 日期时间的比较	55
6.2 逻辑查询	55
6.2.1 使用AND运算符查询同时满足多个条件的记录	55
6.2.2 使用OR运算符查询满足任一条件的记录	56
6.2.3 使用NOT运算符查询满足相反条件的记录	57
6.2.4 复杂逻辑查询	58
6.3 空值查询	58
6.4 使用LIKE操作符实现模糊查询	59
6.4.1 匹配任意单个字符	59
6.4.2 匹配0个或者多个字符	60
6.4.3 使用转义字符	61
6.5 使用REGEXP关键字进行模式匹配	61
6.6 小结	63



第7章 表中数据的排序与分组 .....	64
7.1 使用ORDER BY 子句对数据记录进行排序 .....	64
7.1.1 指定表中的一列进行排序 .....	64
7.1.2 指定表中列的位置序号进行排序 .....	66
7.1.3 对SELECT语句中的非选择列进行排序 .....	66
7.1.4 指定表中的多列进行排序 .....	67
7.2 常用的聚合函数 .....	68
7.3 使用GROUP BY子句对表中数据进行分组 .....	70
7.3.1 单列分组 .....	70
7.3.2 多列分组 .....	71
7.3.3 使用HAVING限制分组后的查询结果 .....	72
7.3.4 对分组结果进行排序 .....	73
7.3.5 GROUP BY子句中处理NULL值 .....	73
7.4 使用ROLLUP关键字统计数据 .....	74
7.5 限制结果集行数 .....	75
7.5.1 使用MySQL数据库限制结果集行数 .....	76
7.5.2 使用Oracle数据库限制结果集行数 .....	77
7.5.3 使用Microsoft SQL Server数据库限制结果集行数 .....	78
7.6 小结 .....	79
第8章 连接查询与集合查询 .....	80
8.1 内连接查询 .....	80
8.1.1 等值连接 .....	80
8.1.2 非等值连接 .....	83
8.1.3 使用ON子句建立相等连接 .....	84
8.1.4 使用USING子句建立相等连接 .....	84
8.2 交叉连接 .....	85
8.3 自连接查询 .....	86
8.4 外连接查询 .....	87
8.4.1 左外连接 .....	88
8.4.2 右外连接 .....	89
8.4.3 全外连接 .....	91
8.5 集合查询 .....	92
8.5.1 并操作 .....	92
8.5.2 交操作 .....	93
8.5.3 差操作 .....	94
8.6 小结 .....	96
第9章 子查询 .....	97
9.1 单行子查询 .....	97
9.2 多行子查询 .....	98
9.2.1 使用IN运算符的子查询 .....	98

9.2.2 使用ANY运算符的子查询 .....	100
9.2.3 使用ALL运算符的子查询 .....	102
9.3 多列子查询 .....	103
9.4 相关子查询 .....	105
9.4.1 带有EXISTS关键字的相关子查询 .....	105
9.4.2 带有NOT EXISTS关键字的相关子查询 .....	107
9.5 在SQL语句中使用子查询 .....	108
9.5.1 在SELECT子句中使用子查询 .....	108
9.5.2 在FROM子句中使用子查询 .....	109
9.5.3 在HAVING子句中使用子查询 .....	110
9.6 多重子查询 .....	111
9.7 在CREATE TABLE语句中使用子查询实现数据表的复制 .....	113
9.8 小结 .....	115
第10章 常用函数 .....	116
10.1 字符函数 .....	116
10.1.1 计算字符串长度 .....	116
10.1.2 将字符串全部转换为小写 .....	118
10.1.3 将字符串全部转换为大写 .....	118
10.1.4 将字符串中单词的首字母转换为大写 .....	118
10.1.5 截取字符串 .....	119
10.1.6 从指定字符串的左侧读取子串 .....	122
10.1.7 从指定字符串的右侧读取子串 .....	122
10.1.8 去除字符串左侧空格或者字符 .....	123
10.1.9 去除字符串右侧空格或者字符 .....	123
10.1.10 去除字符串两侧空格或者字符 .....	124
10.1.11 左侧填充空格或者字符 .....	126
10.1.12 右侧填充空格或者字符 .....	127
10.1.13 取得指定的子串在字符串中的位置 .....	129
10.1.14 颠倒指定字符串的顺序 .....	133
10.1.15 替换指定的子串 .....	134
10.1.16 字符替换 .....	135
10.1.17 拼接字符串 .....	136
10.1.18 取得字符的ASCII码 .....	138
10.1.19 将ASCII码转换为相应的字符 .....	139
10.1.20 匹配发音 .....	139
10.1.21 将字符串重复指定次数 .....	140
10.2 数字函数 .....	141
10.2.1 求绝对值 .....	141
10.2.2 求平方 .....	142
10.2.3 求平方根 .....	142

10.2.4 求对数 .....	142
10.2.5 求幂 .....	144
10.2.6 对指定值进行四舍五入操作 .....	145
10.2.7 求两数相除的余数 .....	146
10.2.8 取得大于等于指定数的最小整数 .....	147
10.2.9 取得小于等于指定数的最大整数 .....	147
10.2.10 求正弦与余弦值 .....	147
10.2.11 求正切值与余切值 .....	148
10.2.12 求反正弦和反余弦值 .....	149
10.2.13 求反正切值 .....	150
10.2.14 弧度与角度的互换 .....	150
10.2.15 取得指定值的符号标志 .....	151
10.2.16 对指定值进行截取操作 .....	152
10.3 日期时间函数 .....	152
10.3.1 取得当前系统的日期和时间 .....	153
10.3.2 对日期值进行加减运算 .....	154
10.3.3 取得日期之后指定工作日对应的日期 .....	159
10.3.4 取得日期值中的指定内容 .....	160
10.3.5 取得指定日期所在月的最后一天 .....	164
10.3.6 取得两个指定月份的差 .....	164
10.3.7 对日期时间进行舍入操作 .....	165
10.3.8 截断指定的日期时间 .....	165
10.4 类型转换函数 .....	166
10.4.1 字符转换函数 .....	166
10.4.2 日期转换函数 .....	170
10.4.3 数值转换函数 .....	175
10.5 比较函数 .....	176
10.5.1 求集合中的最小值 .....	176
10.5.2 求集合中的最大值 .....	176
10.5.3 比较两个字符串 .....	177
10.6 空值处理函数 .....	177
10.6.1 NVL函数与IFNULL函数 .....	177
10.6.2 NVL2函数 .....	179
10.6.3 ISNULL函数 .....	180
10.6.4 COALESCE函数 .....	180
10.7 分支函数与条件表达式 .....	181
10.7.1 IF函数 .....	181
10.7.2 DECODE函数 .....	182
10.7.3 CASE条件表达式 .....	183
10.8 小结 .....	186

第11章 视图的创建与删除 .....	187
11.1 视图的作用 .....	187
11.2 创建视图 .....	187
11.2.1 基于单表创建视图 .....	188
11.2.2 基于多表连接创建视图 .....	189
11.2.3 基于函数、分组数据创建视图 .....	190
11.2.4 为视图添加CHECK约束 .....	191
11.2.5 基于一个已有视图创建新的视图 .....	192
11.2.6 创建只读视图 .....	193
11.3 删除视图 .....	194
11.4 小结 .....	194

## 第四篇 数据更新

第12章 插入数据记录 .....	195
12.1 向数据表中插入数据记录 .....	195
12.1.1 插入单行数据记录 .....	195
12.1.2 向定义有外键约束的表中插入数据记录 .....	197
12.1.3 使用子查询插入多行数据实现表中数据的复制 .....	198
12.1.4 利用MySQL 5.0数据库一次插入多条数据记录 .....	199
12.2 向视图中插入数据记录 .....	200
12.3 小结 .....	201
第13章 修改数据记录 .....	202
13.1 在数据表中修改数据记录 .....	202
13.1.1 修改单行数据记录 .....	202
13.1.2 在定义有外键约束的表中修改数据记录 .....	203
13.1.3 修改多行记录 .....	205
13.1.4 使用子查询修改数据记录 .....	205
13.1.5 使用CASE条件表达式修改多行记录 .....	206
13.1.6 利用MySQL 5.0数据库一次修改多条数据记录 .....	207
13.2 在视图中修改数据记录 .....	208
13.3 小结 .....	209
第14章 删除数据记录 .....	210
14.1 使用DELETE语句删除数据记录 .....	210
14.1.1 删除满足条件的数据记录 .....	210
14.1.2 在定义有外键约束的表中删除数据记录 .....	211
14.1.3 使用子查询删除指定条件的数据记录 .....	212
14.1.4 利用MySQL 5.0数据库一次删除多条数据记录 .....	213
14.1.5 删除数据表中所有记录 .....	213
14.2 使用TRUNCATE语句删除数据表中所有记录 .....	213

14.3 在视图中删除数据记录 .....	214
14.4 小结 .....	215

## 第五篇 数据控制

第15章 权限的授予与回收 .....	217
15.1 数据库及其不同对象允许的操作权限 .....	217
15.2 授予权限 .....	218
15.2.1 授予指定用户操作数据表的权限 .....	218
15.2.2 授予指定用户操作数据列的权限 .....	219
15.2.3 授予指定用户授权的权限 .....	219
15.2.4 授予创建数据表的权限 .....	220
15.2.5 将操作权限授予所有用户 .....	220
15.2.6 使用Administrator管理系统授予用户权限 .....	220
15.3 回收权限 .....	222
15.4 小结 .....	222
第16章 事务的控制与管理 .....	223
16.1 事务的概念 .....	223
16.1.1 原子性 .....	223
16.1.2 一致性 .....	224
16.1.3 隔离性 .....	224
16.1.4 持久性 .....	224
16.2 控制事务 .....	224
16.2.1 开始事务 .....	224
16.2.2 提交事务 .....	225
16.2.3 回滚事务 .....	226
16.3 事务的并发控制 .....	227
16.3.1 并发事务的工作流程 .....	227
16.3.2 事务并发处理中存在的问题 .....	229
16.3.3 事务的隔离级别 .....	230
16.3.4 在数据库中设置隔离级别 .....	230
16.3.5 设置事务访问模式 .....	231
16.4 小结 .....	232

## 第六篇 PL/SQL

第17章 PL/SQL概述 .....	233
17.1 PL/SQL介绍 .....	233
17.1.1 什么是PL/SQL .....	233
17.1.2 为什么要使用PL/SQL .....	234



17.2 一个PL/SQL的例子 .....	235
17.3 PL/SQL编写规范 .....	236
17.3.1 代码注释 .....	236
17.3.2 标识符命名 .....	236
17.3.3 字母大小写 .....	236
17.3.4 代码缩进格式 .....	237
17.4 PL/SQL开发工具简介 .....	237
17.4.1 TOAD .....	237
17.4.2 Navicat .....	238
17.4.3 PL/SQL Developer .....	239
17.5 小结 .....	241
第18章 PL/SQL基础 .....	242
18.1 PL/SQL的块结构 .....	242
18.1.1 基本语句块 .....	242
18.1.2 匿名语句块 .....	244
18.1.3 命名语句块 .....	245
18.1.4 子程序 .....	246
18.1.5 包 .....	248
18.1.6 触发器 .....	249
18.2 PL/SQL基本要素 .....	250
18.2.1 标识符 .....	250
18.2.2 分隔符 .....	251
18.2.3 文字 .....	251
18.2.4 注释 .....	252
18.3 声明和初始化变量 .....	253
18.4 PL/SQL数据类型 .....	254
18.4.1 标量类型 .....	255
18.4.2 复合类型 .....	259
18.4.3 引用类型 .....	260
18.4.4 LOB类型 .....	260
18.5 定义和使用标量变量 .....	260
18.5.1 定义标量变量 .....	260
18.5.2 使用标量变量 .....	261
18.5.3 使用%TYPE属性绑定变量 .....	261
18.6 定义和使用复合类型 .....	262
18.6.1 定义和使用记录 .....	262
18.6.2 定义和使用表 .....	264
18.6.3 定义和使用嵌套表 .....	265
18.6.4 定义和使用变长数组 .....	268
18.7 定义和使用子类型 .....	269

18.7.1 定义子类型 .....	269
18.7.2 使用子类型 .....	270
18.8 变量的作用域 .....	270
18.9 数据类型之间的相互转换 .....	272
18.9.1 隐式数据类型转换 .....	272
18.9.2 显式数据类型转换 .....	272
18.10 小结 .....	273
第19章 PL/SQL中的控制结构 .....	274
19.1 分支控制 .....	274
19.1.1 IF-THEN简单条件语句 .....	274
19.1.2 IF-THEN-ELSE条件分支语句 .....	275
19.1.3 IF-THEN-ELSEIF多重条件分支语句 .....	276
19.1.4 嵌套的IF语句 .....	278
19.2 CASE语句 .....	279
19.2.1 实现等值比较的CASE语句 .....	280
19.2.2 设定标记的CASE语句 .....	282
19.2.3 搜寻式CASE语句 .....	283
19.3 循环控制 .....	284
19.3.1 LOOP循环语句 .....	284
19.3.2 WHILE -LOOP循环语句 .....	286
19.3.3 FOR-LOOP循环语句 .....	287
19.3.4 嵌套循环 .....	288
19.4 顺序控制 .....	288
19.4.1 GOTO语句 .....	288
19.4.2 NULL语句 .....	291
19.5 小结 .....	293
第20章 使用游标 .....	294
20.1 什么是游标 .....	294
20.2 使用显式游标 .....	294
20.2.1 声明游标 .....	295
20.2.2 打开游标 .....	296
20.2.3 从游标中取得结果 .....	297
20.2.4 关闭游标 .....	297
20.2.5 使用BULK COLLECT子句批量绑定数据 .....	298
20.2.6 在游标中使用子查询 .....	299
20.3 游标属性 .....	299
20.3.1 显式游标属性 .....	299
20.3.2 隐式游标属性 .....	302
20.3.3 显式游标属性与隐式游标属性比较 .....	303
20.4 游标循环 .....	303

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

20.4.1 简单LOOP循环	303
20.4.2 WHILE循环	306
20.4.3 游标FOR循环	307
20.4.4 使用游标修改和删除数据行	308
20.4.5 参数化游标	310
20.5 使用游标变量	312
20.5.1 声明游标变量	312
20.5.2 打开游标变量	313
20.5.3 取得数据结果	313
20.5.4 关闭游标变量	314
20.5.5 一个使用游标变量的例子	314
20.5.6 使用游标变量的一些限制	315
20.6 嵌套游标	316
20.7 小结	317
第21章 异常处理	318
21.1 为什么要使用异常处理	318
21.2 声明异常	320
21.3 抛出异常	321
21.4 捕获和处理异常	322
21.4.1 捕获和处理异常的语法规则	322
21.4.2 处理预定义异常	323
21.4.3 处理自定义异常	326
21.4.4 使用内置函数处理异常	327
21.4.5 编译提示EXCEPTION_INIT	329
21.5 异常处理机制	330
21.5.1 声明部分中异常的处理机制	330
21.5.2 可执行部分中异常的处理机制	331
21.5.3 异常处理部分中异常的处理机制	332
21.6 使用异常处理原则	334
21.6.1 对捕获的异常要进行处理	334
21.6.2 确保没有未处理的异常	334
21.6.3 标记异常发生的位置	335
21.6.4 控制异常处理程序	337
21.7 小结	338
第22章 存储过程	339
22.1 创建存储过程	339
22.2 参数模式	341
22.2.1 IN模式	341
22.2.2 OUT模式	344
22.2.3 IN OUT模式	347

22.3 调用存储过程 .....	348
22.3.1 调用无参数的存储过程 .....	348
22.3.2 调用带有输入参数的存储过程 .....	348
22.3.3 调用带有输出参数的存储过程 .....	350
22.3.4 调用带有输入输出参数的存储过程 .....	351
22.4 参数传递 .....	352
22.4.1 使用参数位置传递参数值 .....	352
22.4.2 使用参数名称传递参数值 .....	352
22.4.3 使用位置和名称传递参数值 .....	353
22.4.4 使用NOCOPY编译提示传递参数 .....	353
22.4.5 使用参数的默认值 .....	355
22.5 删除存储过程 .....	356
22.6 小结 .....	356
第23章 函数 .....	357
23.1 创建函数 .....	357
23.1.1 创建函数的语法规则 .....	357
23.1.2 函数体中的RETURN子句 .....	358
23.1.3 创建有输入参数的函数 .....	359
23.1.4 创建有输出参数的函数 .....	360
23.1.5 创建有输入输出参数的函数 .....	361
23.1.6 创建记录类型作为返回值的函数 .....	362
23.1.7 创建集合类型作为返回值的函数 .....	362
23.2 调用函数 .....	363
23.2.1 调用无参数的函数 .....	363
23.2.2 调用带有输入参数的函数 .....	364
23.2.3 调用带有输出参数的函数 .....	365
23.2.4 调用带有输入输出参数的函数 .....	365
23.2.5 调用返回值为记录类型的函数 .....	365
23.2.6 调用返回值为集合类型的函数 .....	366
23.3 参数传递 .....	367
23.3.1 使用参数位置传递参数值 .....	367
23.3.2 使用参数名称传递参数值 .....	367
23.3.3 使用参数名称和位置传递参数值 .....	367
23.4 函数在SQL中的应用 .....	368
23.4.1 纯度规则 .....	368
23.4.2 在SQL中调用函数 .....	369
23.5 删除函数 .....	370
23.6 存储过程与函数 .....	370
23.7 小结 .....	371

第24章 包 .....	372
24.1 创建包 .....	372
24.1.1 创建包说明 .....	372
24.1.2 创建包体 .....	373
24.2 调用包中的公有元素 .....	376
24.3 在包中使用重载 .....	377
24.4 删除包 .....	379
24.5 系统包 .....	380
24.5.1 生成并发送报警信息的包DBMS_ALERT .....	380
24.5.2 输入和输出信息的包DBMS_OUTPUT .....	382
24.5.3 进行管道通信的包DBMS_PIPE .....	383
24.5.4 安排和管理PL/SQL语句块的包DBMS_JOB .....	387
24.5.5 处理LOB数据类型数据的包DBMS_LOB .....	389
24.6 小结 .....	393
第25章 触发器 .....	394
25.1 触发器简介 .....	394
25.1.1 触发器的类型 .....	394
25.1.2 触发器的用途 .....	395
25.1.3 触发器的限制 .....	395
25.2 创建DML触发器 .....	396
25.2.1 创建语句触发器 .....	396
25.2.2 创建行触发器 .....	398
25.2.3 触发器谓词 .....	400
25.3 创建DDL触发器 .....	401
25.4 创建INSTEAD OF触发器 .....	401
25.5 创建事件触发器 .....	402
25.5.1 事件属性函数 .....	403
25.5.2 创建系统事件触发器 .....	405
25.5.3 创建用户事件触发器 .....	407
25.6 管理触发器 .....	407
25.6.1 禁用触发器 .....	407
25.6.2 激活触发器 .....	408
25.6.3 重新编译触发器 .....	408
25.6.4 删除触发器 .....	408
25.7 小结 .....	409

## 第七篇 SQL应用

第26章 SQL语句性能优化 .....	411
26.1 适当创建索引 .....	411



26.2 优化查询语句 .....	412
26.2.1 避免在SELECT语句中使用“*” .....	412
26.2.2 调整WHERE子句中连接条件的顺序 .....	412
26.2.3 避免使用OR关键字 .....	413
26.2.4 避免使用<>和!= 操作符 .....	414
26.2.5 相关子查询中使用EXISTS关键字代替IN关键字 .....	414
26.2.6 使用LIKE关键字 .....	415
26.2.7 避免使用HAVING子句 .....	416
26.2.8 使用存储过程 .....	416
26.3 规范SQL语句书写格式 .....	416
26.4 小结 .....	417
第27章 动态SQL .....	418
27.1 使用EXECUTE IMMEDIATE语句处理单行查询 .....	418
27.2 使用游标变量处理多行查询 .....	420
27.2.1 定义游标变量 .....	420
27.2.2 打开游标变量 .....	420
27.2.3 从游标变量取得数据 .....	421
27.2.4 关闭游标变量 .....	422
27.2.5 动态处理多行查询 .....	422
27.3 批量绑定 .....	423
27.3.1 使用FORALL语句批量绑定 .....	423
27.3.2 使用EXECUTE IMMEDIATE语句批量绑定 .....	424
27.3.3 使用FETCH 语句批量绑定 .....	425
27.4 使用DBMS_SQL包实现动态SQL .....	426
27.4.1 DBMS_SQL包中常用存储过程和函数 .....	426
27.4.2 使用DBMS_SQL包实现动态SQL的查询操作 .....	427
27.4.3 使用DBMS_SQL包实现动态SQL的DML操作 .....	428
27.4.4 使用DBMS_SQL包实现动态SQL的DDL操作 .....	429
27.5 小结 .....	430
第28章 数据库的存取访问 .....	431
28.1 数据库应用系统结构 .....	431
28.1.1 集中式数据库系统 .....	431
28.1.2 客户端/服务器端数据库系统 .....	432
28.1.3 并行式数据库系统 .....	432
28.1.4 分布式数据库系统 .....	433
28.2 数据库连接访问 .....	433
28.2.1 ODBC .....	433
28.2.2 OLE DB .....	433
28.2.3 ADO .....	434
28.2.4 JDBC .....	434

## 零基础学SQL

28.3 Java与MySQL 5.0数据库连接与访问 .....	435
28.3.1 Java开发环境 .....	435
28.3.2 安装Eclipse .....	437
28.3.3 添加MySQL 5.0驱动程序 .....	439
28.3.4 Java与数据库的连接过程 .....	439
28.4 Java与MySQL 5.0数据库开发 .....	440
28.4.1 查询数据 .....	440
28.4.2 更新数据 .....	442
28.5 小结 .....	444
附录A 常用SQL语句 .....	445
附录B 常用函数对照 .....	450

# 第一篇

## 关系数据库与SQL语言

### 第1章 关系数据库介绍

关系数据库是一个二维表的集合，可以用来存储不同类型的数据信息。用户可以根据自己的需要查询其中的信息。目前经常使用的数据库包括Oracle数据库、MySQL数据库、Microsoft SQL Server数据库和DB2数据库等。本章主要介绍关系数据库中涉及的几个重要的概念以及几种常用的关系数据库，最后以MySQL 5.0数据库为例，介绍了MySQL 5.0数据库的安装和使用方法。

本章重点：

- ☐ 概念模型和关系数据模型
- ☐ 关系模式
- ☐ 常用关系数据库介绍
- ☐ MySQL 5.0数据库的安装和使用方法

#### 1.1 数据模型

数据模型主要包括概念模型、逻辑数据模型和物理数据模型。概念模型是以客户的观点和想法为基础，对现实世界事物的抽象；逻辑数据模型是指用户看到的数据库中的数据模型，常用的是关系数据模型；物理数据模型是用来表示数据的存储结构的。本节主要介绍概念模型和关系数据模型。

##### 1.1.1 概念模型

在关系数据库的设计中，概念模型通常是通过E-R图来描述的。其中，E表示实体的意思；R表示关系的意思。因此E-R图也叫做实体—关系图。

在E-R图中的E是英文单词Entity的缩写，表示实体的意思。这里所说的实体可以理解为现实世界中的事物，例如，高等院校中的院系、教师等。E-R图中的R是英文单词Relationship的缩写，表示关系的意思。这里所说的关系可以理解为实体与实体之间的相互联系。例如，高等院校中院系与教师之间的相互联系。在E-R图中还涉及的一个概念是属性，英文单词为Attribute，它用来描述实体的特征。例如，高等院校中院系的编号、名称；教师的姓名、编号、工资、所在院系等。

在E-R图中，关系是用来表示实体与实体之间相互联系的。关系可以分为一对一、一对多和多对多

## 零基础学SQL

三种类型。下面通过例子来讲解关系中的这3种类型。

- 一对一 (1:1)：在高等院校中，校长和学校的关系就是一对一的关系。每一个学校只有一名校长，一名校长只能管理一个学校。
- 一对多 (1:n)：在高等院校中，院系和学生之间就是一对多的关系。一个院系中可以对应多个学生，而每一个学生只是其中某一个院系中的成员。
- 多对多 (n:m)：在高等院校中，课程与授课教师之间就是多对多关系。一门课程可以由几个不同的教师来讲授，一名教师也可以讲授多门不同的课程。

在E-R图共有3种符号：矩形、椭圆（或者圆形）和菱形。其中，矩形用来表示实体，椭圆或者圆形用来表示属性，菱形用来表示关系。下面来看一下如何使用E-R图描述上面讲到的3种关系。

### 1. 使用E-R图描述一对一关系

这里以学校和校长为例，使用E-R图描述一对一关系。其E-R图如图1.1所示。

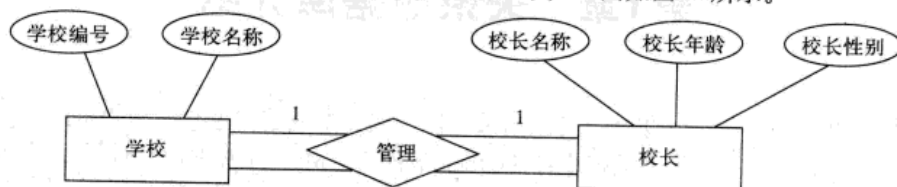


图1.1 一对一关系

图1.1描述了学校和校长的一对一关系。其中，矩形中的学校和校长表示实体；椭圆中的学校编号、学校名称表示实体学校的属性，校长名称、校长年龄、校长性别表示实体校长的属性；菱形中的管理表示学校和校长之间的关系，即学校是由校长来管理的。

### 2. 使用E-R图描述一对多关系

这里以学生和院系为例，使用E-R图描述一对多关系。其E-R图如图1.2所示。

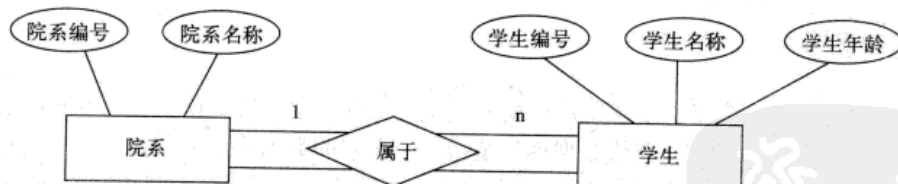


图1.2 一对多关系

图1.2描述了学生和院系的一对多关系。其中，矩形中的学生和院系表示实体；椭圆中的院系编号、院系名称表示实体院系的属性，学生编号、学生名称、学生年龄表示实体学生的属性；菱形中的属于表示院系和学生之间的关系，即学生是属于某一个院系的。

### 3. 使用E-R图描述多对多关系

这里以教师和课程为例，使用E-R图描述多对多关系。其E-R图如图1.3所示。

图1.3描述了教师和多门课程的多对多关系。其中，矩形中的教师和课程表示实体；椭圆中的课程编号、课程名称表示实体课程的属性，教师编号、教师名称、教师职称表示实体教师的属性；菱形中的授课表示院系和学生之间的关系，即课程是由教师来讲授的。

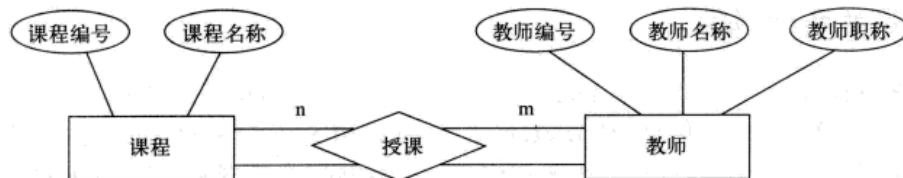


图1.3 多对多关系

### 1.1.2 关系数据模型

在数据库管理系统的实现中，关系数据模型通过二维表的形式描述实体与属性之间的关系。两个二维表之间的关系也包括一对一、一对多和多对多3种类型。

- 一对一 (1:1)：在二维表A和二维表B中，如果表A中的一条记录只与表B中的一条记录相对应，同时表B中的每一条记录也只与表A中的一条记录相对应，那么这两个数据表之间就是一对一的关系。
- 一对多 (1:n)：二维表A和二维表B中，如果表A中的一条记录只与表B中的一条记录相对应，但是表B中的每一条记录却能对应表A中的多条记录，那么这两个数据表之间就是一对多的关系。
- 多对多 (n:m)：二维表A和二维表B中，如果表A中的一条记录与表B中的多条记录相对应，同时表B中的每一条记录也与表A中的多条记录相对应，那么这两个数据表之间就是多对多的关系。

二维表是由行和列两部分组成的。其中，二维表中的行用来描述实体中的具体数据，二维表中的列用来表示实体中的属性。在二维表还涉及与其有关的一些概念，这些与二维表有关的概念将在4.1节中介绍。

## 1.2 关系模式

关系模式是用来表示对关系的描述的。关系数据库中共有3种关系模式：概念模式、外模式和内模式。下面就对关系数据库中的3种关系模式做一个简单的介绍。

- 概念模式 (Conceptual Schema)：概念模式也叫做模式，用来描述数据库中的数据逻辑结构。可以把概念模式理解为现实世界中的实体在数据库系统中的具体实现。例如，一个关系逻辑结构对应的一个二维数据表，就可以理解为关系数据库中的模式。概念模式与应用程序和计算机硬件等环境无关。
- 外模式 (External Schema)：外模式也叫做用户模式，它是概念模式的一部分。在外模式中定义了允许用户操作的数据，例如，在数据库中用户看到的视图就可以理解为关系数据库中的外模式。当然，在关系数据库中由于用户的需求、存储数据的不同，不同数据的外模式也不是完全相同的。
- 内模式 (Internal Schema)：内模式也叫做存储模式，它是用来描述数据的物理结构和数据的存储方式的。例如，关系数据库中索引的组织方式、数据记录的存储方式等就可以理解为关系数据库的内模式。



## 1.3 常用关系数据库

目前主流的数据库厂商主要包括Oracle、Microsoft、Sybase、Informix、IBM等，每一个厂商都有自己不同版本的数据库产品。例如，Oracle数据库、Microsoft SQL Server数据库、Microsoft Access数据库、FoxPro数据库等。另外还有一些厂商提供了一些开源版本的数据库产品，主要包括MySQL数据库、PostgreSQL数据库、SAP等。这些数据库都是目前比较常用的关系数据库。本节将介绍几种主要的常用关系数据库。

### 1.3.1 Oracle数据库

Oracle数据库是美国Oracle公司（甲骨文）推出的关系数据库系统，它是目前主流的广泛使用的数据库系统之一。作为大型的数据库系统，Oracle数据库提供了完整的数据管理功能，主要应用于大、中型应用系统、C/S（客户端/服务器）、B/S（浏览器/服务器）系统中的服务器端。对于数据量大、并发操作多、实时性要求高的系统，服务器端的数据库一般都选择Oracle数据库。

Oracle公司从1984年推出运行在PC机上的Oracle数据库到现在，版本在不断地变化和更新。在1986年的第5版中，增加了Oracle数据库的分布式处理机制；在1992年的第7版中，提供了比较完善的分布式数据库功能；到2001年推出的Oracle9i是一个完整的、简单的、用于互联网的、智能化、安全可靠的数据库产品；随后Oracle公司又推出了Oracle10g，在Oracle10g中加入了网格计算的功能，并在安全性、可伸缩性、可用性等方面都得到了加强，并提供了SQL语言和PL/SQL语言对正则表达式的支持；在2007年又推出了Oracle 11g，该版本中在数据库管理和PL/SQL部分增加了许多新的特性。

### 1.3.2 Microsoft SQL Server数据库

Microsoft SQL Server数据库是Microsoft公司（微软）推出的关系数据库系，也是目前主流的广泛使用的数据库系统之一。SQL Server数据库具有高性能、可扩展、先进的系统管理、支持Windows图形化管理工具、杰出的事务处理功能等特点。

Microsoft SQL Server数据库的1.0版本是在1989年推出的，之后经历了1.11、SQL Server for Windows NT 3.1、6.0、7.0等版本。在2000年、2005年和2008年Microsoft公司分别推出了SQL Server 2000、SQL Server 2005和SQL Server 2008这3个版本。目前常用的两个版本是SQL Server 2005和SQL Server 2008。SQL Server 2008中在安全性、可靠性、可扩展性等方面都比SQL Server 2005有了很大的改进，同时还增加了T-SQL语言的功能。例如，增加了T-SQL的行构造器、增加了日期和时间的数据类型、增加了MERGE语句等。

### 1.3.3 MySQL数据库

MySQL是瑞典的MySQL AB公司开发的一款功能强大、使用灵活、多用户、多线程SQL的数据库管理系统。为用户提供了丰富的应用程序接口和非常有用的功能集，是互联网中流行的数据库服务器，很多软件开发人员和商业用户也都在使用MySQL数据库。

MySQL数据库是由C和C++语言编写的，它支持多线程，为不同的编程语言像Perl、PHP、Java、Python、C++语言等都提供了相应的API，并且具有操作简单、性能高、可移植性好、安装时占用的资源少等特点。而它的最大特点就是对于个人用户它是免费的，可以到其官方网站<http://www.mysql.com/>下载，其中文网址为<http://www.mysql.cn/>。目前MySQL数据库的最新版本是MySQL 5.0，本书中使用

的就是这个版本。

### 1.3.4 PostgreSQL数据库

PostgreSQL数据库是以Postgres 版本4.2为基础，由美国加州伯克利分校开发的一款对象关系型数据库管理系统。PostgreSQL数据库采用的比较经典的 C/S（客户/服务器）结构，它支持事务、存储过程、并发控制，性能优异。PostgreSQL数据库服务器还提供了统一的客户端 C 接口，像ODBC、JDBC、Perl、C/C++等不同的客户端接口都是源自这个C接口，而且PostgreSQL数据库几乎支持所有类型的数据库客户端接口。PostgreSQL数据库还拥有极其强大的扩展能力，可以很容易地扩展数据类型、函数、操作符、索引方法等。

与MySQL数据库相比，PostgreSQL数据库提供的功能要比MySQL数据库丰富，但是在速度和稳定性方面它不及MySQL数据库。PostgreSQL数据库也是免费的，可以到其官方网站<http://www.postgresql.com/>下载。

## 1.4 安装与使用MySQL 5.0数据库

MySQL是一款功能强大、使用灵活、多用户、多线程SQL的数据库管理系统。它具有操作简单、性能高、可移植性好、安装时占用的资源少等特点。这一节将介绍MySQL 5.0和用户图形界面的安装以及MySQL 5.0的运行过程。

### 1.4.1 安装MySQL 5.0

MySQL 5.0的最新版本可以到其官方网站<http://www.mysql.com/>下载，也可以到其中文网站下载，其中文网站的网址为<http://www.mysql.cn/>。完成下载后，就可以安装MySQL数据库。这里以MySQL 5.0为例，来介绍如何安装MySQL 5.0数据库。

(1) 将下载的文件解压缩后，执行安装文件，出现安装向导界面，如图1.4所示，单击“Next”按钮。

(2) 在弹出的“Setup Type”界面中，选择“Typical”安装类型，单击“Next”按钮，如图1.5所示。

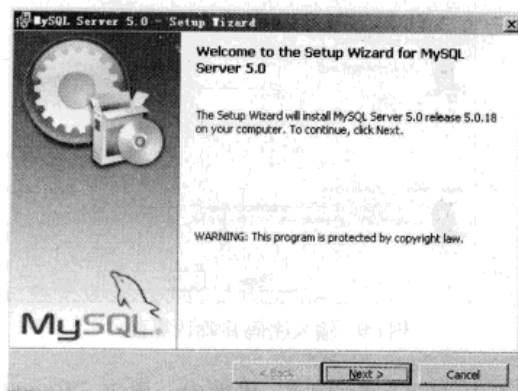


图1.4 MySQL 5.0安装向导界面

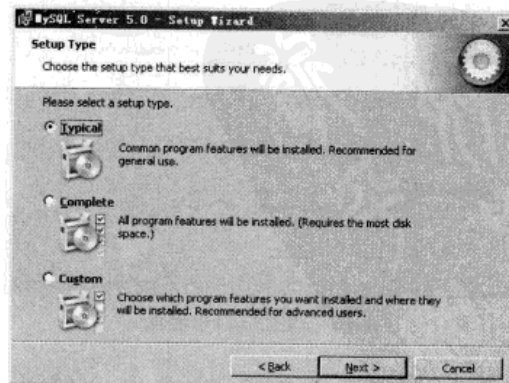


图1.5 选择安装类型界面

(3) 在弹出的“Ready to Install the Program”界面中，单击“Install”按钮，如图1.6所示。

(4) 安装完成后，在出现的“MySQL 5.0.com Sign-up”界面中，选择“skip Sign-up”单选按钮，单击“Next”按钮后，出现如图1.7所示的界面。

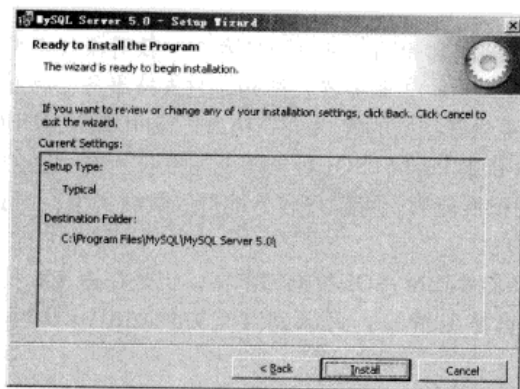


图1.6 安装MySQL 5.0数据库界面

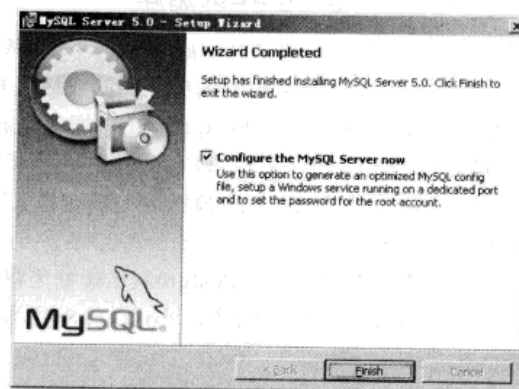


图1.7 安装完成界面

(5) 安装完成后，会出现“Welcome to the MySQL 5.0 server Instance Configuration Wizard 1.0.8”的界面。单击“Finish”按钮。

(6) 然后会出现“MySQL Server Instance Configuration”界面，继续安装MySQL Server。连续单击“Next”按钮，直到出现“Please set the networking options”界面。

(7) 在出现的“Please set the networking options”界面中设置端口号。这里使用默认的端口号3306。如果想修改端口号，可以在“Port Number”所指示的下拉列表中修改，修改完成后单击“Next”按钮，如图1.8所示。

(8) 在出现的“Please set security options”界面中输入密码root，并在确认后，单击“Next”按钮，如图1.9所示。

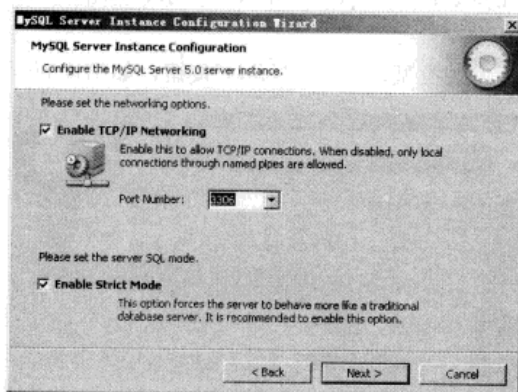


图1.8 设置端口号界面

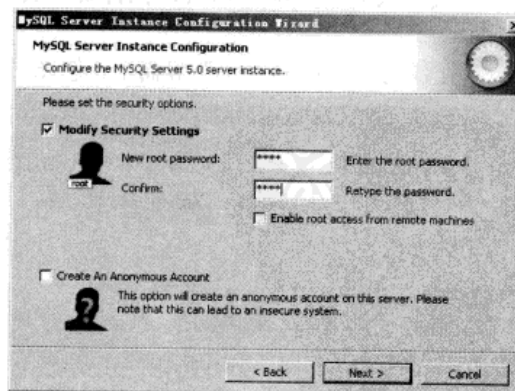


图1.9 输入密码并确认界面

(9) 在出现的“Configuration the MySQL Server 5.0 server instance”界面中，单击“Execute”按钮。

(10) 单击“Exceute”按钮后，会执行MySQL Server的安装程序，出现如图1.10所示的画面。

(11) 正常安装结束后，单击“开始”|“所有程序”|“MySQL 5.0”|“MySQL 5.0 Command Line

Client” 命令。运行后在该界面中输入密码root，其结果如图1.11所示。

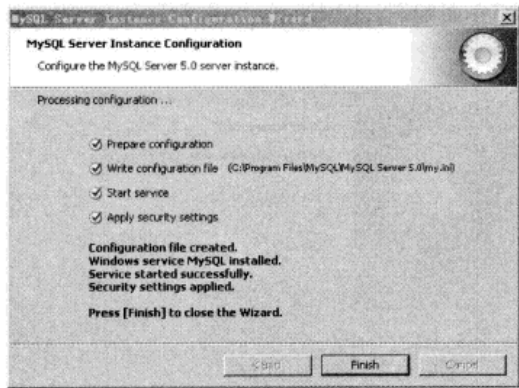


图1.10 安装MySQL Server界面

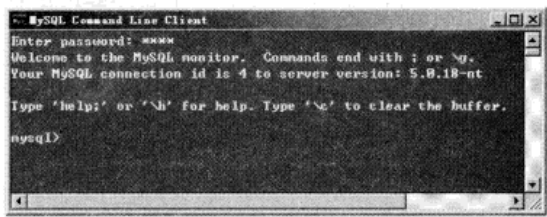


图1.11 MySQL 5.0 Command Line Client运行画面

### 1.4.2 安装用户图形界面

安装完成MySQL 5.0后，就可以进行数据操作了，如图1.11所示，但是其操作环境的界面并不友好，因此需要为其安装一个与用户交互的图形界面。MySQL的用户界面主要有MySQL Administrator和MySQL Query Brower。其中，MySQL Query Brower主要用于编写和调试SQL语句，这里安装的就是MySQL Query Brower。其安装过程如下：

- (1) 执行“mysql-gui-tools-5.0-r10-win32.msi” 文件，会出现如图1.12所示的画面，然后单击“Next” 按钮。
- (2) 在出现的“License Agreement” 界面中，单击“I accept the terms in the license agreement” 后的单选按钮。
- (3) 在出现的“Destination Folder” 界面中，出现选择安装路径的界面，选择默认的安装路径后单击“Next” 按钮，如图1.13所示。

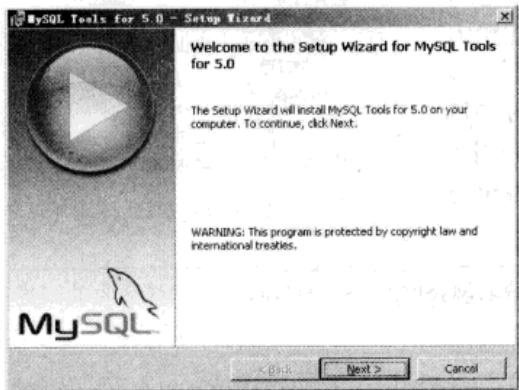


图1.12 用户图形安装界面

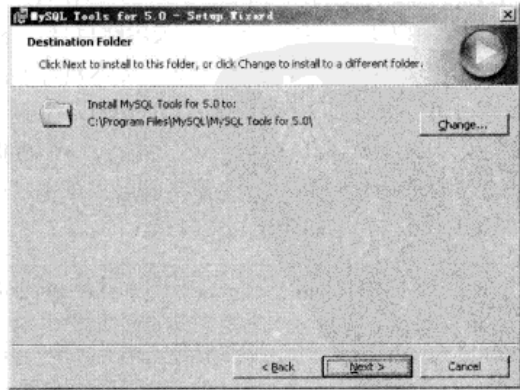


图1.13 选择安装路径界面

- (4) 在出现的“Setup Type” 界面中选择“Complete” 选项。
- (5) 在出现的“Ready to Install the Program” 界面中，单击“Next” 按钮，其安装界面如图1.14

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

零基础学SQL

所示。  
(6) 单击“Install”按钮，开始安装。安装完成后的界面如图1.15所示。单击“Finish”按钮，完成用户图形界面的安装。

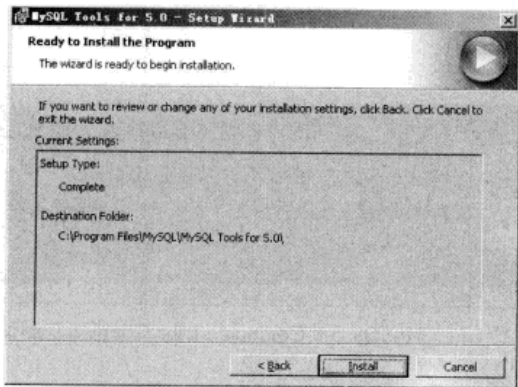


图1.14 用户图形安装界面

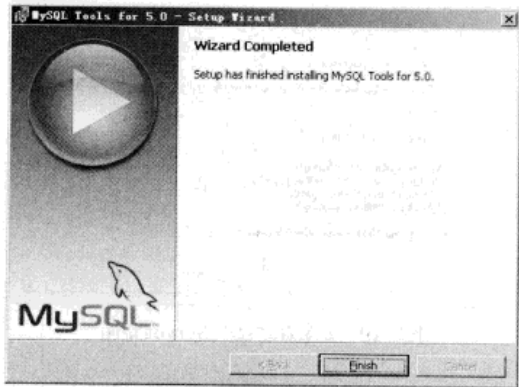


图1.15 安装完成界面

(7) 在安装完成后，选择“开始”|“所有程序”|“MySQL”命令会出现如图1.16所示的内容。

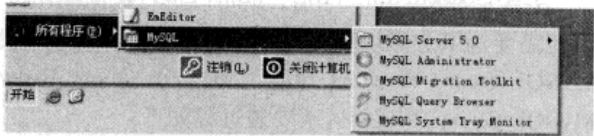


图1.16 MySQL 5.0文件内容界面

1.4.3 运行MySQL 5.0

在安装完成图形界面工具以后，就可以运行MySQL 5.0了。单击“开始”|“MySQL”|“MySQL Query Brower”命令，会出现一个用户确认界面，如图1.17所示。

在第一次进入MySQL 5.0的图形用户界面时，需要填写一些用户信息。在该界面的“Server Host”中需要输入127.0.0.1表示本地机器；“Port”表示端口号，这里使用默认的端口号3306；在“Username”文本框中输入root；在“Password”文本框中输入安装MySQL 5.0时设置的密码root；在“Default Schema”文本框中输入一个数据库的名字。这里输入的是test\_STInfo。设置完成后，单击“OK”按钮进入MySQL 5.0。其界面如图1.18所示。

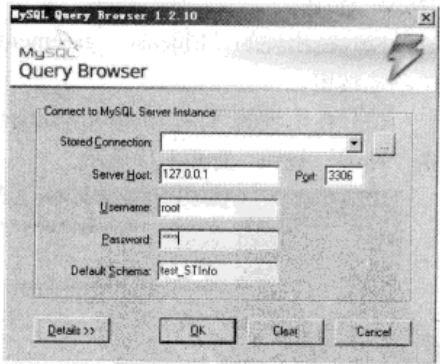


图1.17 用户确认界面

**注意** 如果在“Default Schema”文本框中输入的数据库在MySQL 5.0中不存在，则系统会为其创建一个同名的数据库。

图1.18显示的是一个MySQL 5.0图形用户界面。其中，最上面的标题栏显示的是用户名，连接机器的Server Host、端口号，以及操作的数据库名等信息；其下是菜单栏，菜单栏主要包括File、Edit、View、Query、Script、Tools、Window和Help；在菜单栏下面的空白处用来编写SQL语句；中间的空白处用来显示查询结果；界面左侧在Schemata中有一个test\_STInfo的数据库，这就是用户新创建的数据库，但该数据库中没有任何的表。另外两个数据库是MySQL 5.0自带的数据库。



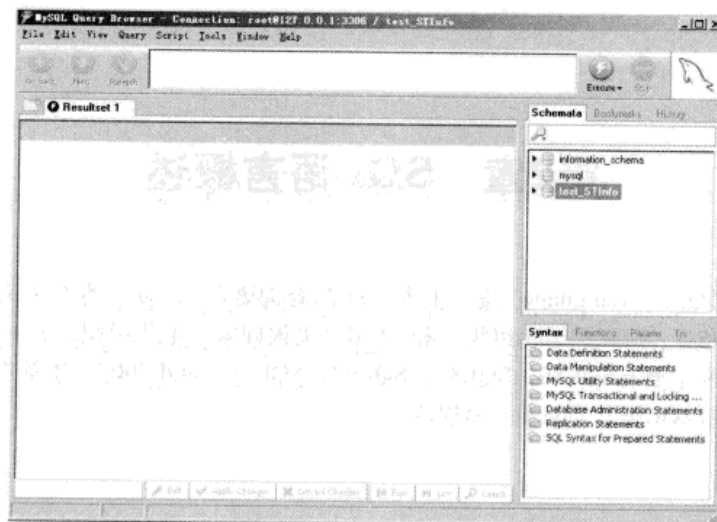


图1.18 MySQL 5.0图形用户界面

## 1.5 小结

本章主要介绍了关系数据库中数据模型中涉及的3个重要概念：概念模型和关系数据模型以及关系模式。另外还介绍了几种常用的数据库，并详细讲解了MySQL 5.0数据库的安装和使用方法。在本书中，将以MySQL 5.0数据库为基础对SQL语言进行讲解。在书中以后出现的SQL语言查询或者更新结果的插图都是在MySQL 5.0数据库下运行的结果。

## 第2章 SQL语言概述

SQL (Structured Query Language) 是一种结构化的查询语言，它是实现与关系数据库通信的标准语言。SQL标准是由ISO（国际标准化组织）和ANSI（美国国家标准化组织）共同制定的，从1983年开始到目前经历的标准主要有SQL86、SQL89、SQL92、SQL99、SQL2003。本章主要介绍SQL语言中的一些基本内容并了解SQL语言的基本书写规范。

本章重点：

- ☐ SQL语言介绍
- ☐ SQL语句的分类
- ☐ SQL语言的特点
- ☐ SQL语言中常用数据类型
- ☐ SQL语句书写规范

### 2.1 SQL语言介绍

SQL作为关系数据库中操作的标准语言，集数据定义语言（Data Definition Language, DDL）、数据查询语言（Data Query Language, DQL）、数据操作语言（Data Manipulation Language, DML）、数据控制语言（Data Control Language, DCL）和事务控制语言的功能于一体。SQL语言主要用于完成对数据库的操作，例如查询、增加、修改、删除数据，创建和删除数据库对象，修改表结构等。

SQL语言可以作为大部分数据库的共同数据存取语言 and 标准接口，其中比较常用的数据库包括Oracle、Microsoft SQL Server、MySQL、PostgreSQL、DB2、Microsoft Access等。这些数据库系统都可以使用SQL语言执行数据的操作。

SQL标准最早是基于IBM的实现由美国国家标准化组织ANSI (American National Standards Institute) 在1986年批准的，1987年国际标准化组织ISO (International Standards Organization) 将其作为国际标准采用。在1992年ISO对其进行了修订，1999年又再次进行了修订，目前最新的标准是SQL2003。

不同厂商的SQL产品及其关系数据库管理系统在SQL的实现上大部分是与SQL标准相兼容的，但是它们也并不是完全按照这个标准来实现，因此不同厂商生产的关系数据库系统在SQL的实现上还是有所差异的。另外，不同厂商生产的关系数据库系统都对SQL做了不同程度的扩展，目前流行的对标准SQL的扩展版本主要有两个：PL/SQL 和Transact-SQL。

- ☐ PL/SQL是基于Oracle数据库的通信语言，Oracle公司已经将PL/SQL语言集成到Oracle的服务器中，它可以运行在任何的Oracle开发环境中。许多厂商也提供了基于Oracle数据库的专用的访问工具，比较常用的包括TOAD、Navicat和PL/SQL Developer。
- ☐ Transact-SQL是基于Sybase数据库与Microsoft SQL Server数据库中的数据库通信语言。其主要

的运行环境是SQL Server Management Studio和SQLCMD。其中SQLCMD是一个命令行脚本工具，作为SQL Server数据库的访问工具，它可以在数据库服务器的任何目录路径下的命令行提示符窗口中执行。

本书中讲解的SQL语言是以SQL92为标准，并兼顾目前主流数据库中在使用SQL语句执行某些操作上的差异。另外也会介绍一些SQL99中新增加的内容，例如BOOLEAN等数据类型、集合操作、触发器创建和操作等。

## 2.2 SQL语句的分类

SQL语句主要包括数据定义语言（DDL）、数据查询语言（DQL）、数据操作语言（DML）、数据控制语言（DCL）和事务控制语言等。

- 数据定义语言（DDL）：主要用于创建、修改和删除数据库对象（数据表、视图、索引等），包括CREATE、ALTER、DROP语句。其中，CREATE语句用于创建数据库对象，例如，CREATE TABLE表示创建数据表，CREATE VIEW表示创建视图，CREATE INDEX表示创建索引；ALTER语句用于修改数据库对象，例如，ALTER TABLE表示修改数据表的结构；DROP语句用于删除数据库对象，例如，DROP TABLE表示删除数据表，DROP VIEW表示删除视图，DROP INDEX表示删除索引。
- 数据查询语言（DQL）：主要用于查询数据库中的数据。其主要语句为SELECT语句。SELECT语句是SQL语言中最重要的部分。SELECT语句中主要包括5个子句，分别是FROM子句、WHERE子句、GROUP BY子句、HAVING子句和WITH子句。
- 数据操作语言（DML）：主要用于更新数据库里数据表中的数据，包括INSERT、UPDATE、DELETE语句。其中，INSERT语句用于向数据库中插入数据；UPDATE语句用于修改数据库中的数据；DELETE语句用于删除数据库中的数据。
- 数据控制语言（DCL）：主要用于授予和回收访问数据库的某种权限。包括GRANT、REVOKE等语句。其中，GRANT语句用于向用户授予权限；REVOKE语句用于向用户回收权限。
- 事务控制语言：主要用于数据库对事务的控制，保证数据库中数据的一致性，包括COMMIT、ROLLBACK等语句。其中，COMMIT用于事务的提交；ROLLBACK用于事务的回滚。

数据定义语言（DDL）、数据查询语言（DQL）、数据操作语言（DML）和数据控制语言（DCL）中涉及的SQL语句在以后的章节中都会介绍，其中，第4章和第11章主要介绍数据定义语言（DDL）在数据表和视图中的应用；第5章～第10章主要介绍数据查询语言（DQL）在查询数据库数据方面的应用；第12章～第14章主要介绍数据操作语言（DML）在更新数据库中的数据方面的应用；第15章和第16章主要介绍数据控制语言（DCL）在授予和回收访问数据库权限以及对事务控制方面的应用。

## 2.3 SQL语言的特点

SQL是关系数据库中操作的标准语言，是一种非过程化语言。在数据库应用开发中，通过使用SQL语言可以完成数据表的创建、授予或者回收用户对数据库的存取权限，对数据进行查询、增加、修改、删除等功能。SQL语言的特点主要体现在以下几个方面。

- 简单易学，语言结构简便。SQL语言中的语句命令较少，而且其语句命令与英文中的自然语言

零基础学SQL

也很相近，语法也并不复杂，比起那些程序设计语言中的语法要简单得多，而且SQL语言也不需要关心运行系统底层的算法实现的过程，因此更容易学习和掌握。

- ❑ 非过程化语言。SQL语言是一种非过程化语言。使用SQL语言进行数据库操作时，开发人员或者用户只需要关心需要做的是做什么，而不需要关心它是如何去执行这样的操作的。例如，要查询某个数据表中的数据记录，开发人员或者用户要做的就是使用SQL语言告诉数据库管理系统要检索什么样的数据记录，从哪一个数据表中检索，而对于这些记录是如何存取的、它们的存取路径是什么等信息都是由数据库管理系统自己来完成的，不需要开发人员或者用户去关心。数据库管理系统会优化访问路径并将执行的结果返回。
- ❑ 采用集合操作方式。无论是执行数据的查询操作，还是执行对数据的增加、修改和删除的更新操作，都可以是集合的操作方式。例如，使用一条SELECT语句可以将满足条件的多条数据记录全部查询出来，查询出来的是符合查询条件的数据记录的集合（读者可以思考一下，使用Java等编程语言，如果想获得一个集合中的记录，应该如何操作）。采用集合操作方式，可以大大提高数据的处理速度。
- ❑ SQL语言能够嵌入到高级语言中。SQL语言可以作为嵌入式语言，嵌入到高级语言（例如C语言等）中执行。SQL语言这种灵活性的使用方式也为程序开发人员开发和设计程序提供了极大的便利。

## 2.4 常用数据类型

在创建数据表时，除了需要创建数据表的表名、列名之外，还需要为数据表中的每一列选择合适的数据类型。在数据库中，常用的数据类型包括整数类型、浮点类型、数值类型、日期与时间类型、字符类型、二进制类型等。不同的数据库中，数据类型的定义也不完全相同。这一节就以Oracle数据库、MySQL数据库和Microsoft SQL Server数据库为例，介绍在这3种数据库中，常用的数据类型定义及其使用方法。

### 2.4.1 整数类型与浮点类型

整数类型的数据可以是正整数，也可以是负整数。在MySQL数据库中主要整数类型及其取值范围如表2.1所示。

表2.1 MySQL中主要整数类型及其取值范围

整数类型	取值范围	整数类型	取值范围
TINYINT	-128 ~ 127 (1个字节)	INT	-2 <sup>31</sup> ~ 2 <sup>31</sup> -1 (4个字节)
SMALLINT	-32768 ~ 32767 (2个字节)	BIGINT	-2 <sup>63</sup> ~ 2 <sup>63</sup> -1 (8个字节)
MEDIUMINT	-2 <sup>23</sup> ~ 2 <sup>23</sup> -1 (3个字节)		

在MySQL数据库中，这些整数类型还可以带一个参数用来表示数据最大的显示宽度。例如INT(4)表示显示数据列的列宽度为4。使用参数并不影响整数类型的取值范围。这个参数是可选的。

在MySQL数据库中，浮点类型主要包括FLOAT和DOUBLE。其中，FLOAT表示单精度浮点数，DOUBLE表示双精度浮点数。浮点数类型及其取值范围如表2.2所示。

表2.2 MySQL中浮点数类型及其取值范围

浮点数类型	取值范围
FLOAT(m,n)	-2.4E38~2.4E38（占4个字节）精确到小数点后7位
DOUBLE(m,n)	-2.7E308~2.7E308（占8个字节）精确到小数点后15位

在MySQL数据库中，这些浮点数类型还可以包括两个参数。其中，参数m表示存储数据的有效数字的位数；参数n表示小数点后的位数。例如FLOAT（5,2），第一个参数5表示显示的数字总位数为5，第二个参数表示小数点后的数字个数。如果把数据122.456插入到定义为FLOAT（5,2）的数据列中，则实际放入到该列的数据为122.46。

可以看到，MySQL数据库中数字在使用参数进行插入时会对插入的数据进行四舍五入的操作。浮点数类型中这两个参数也是可选的。

在Microsoft SQL Server数据库中，也可以使用TINYINT、SMALLINT、INT和BIGINT存储整型数据，其取值范围与表2.1中TINYINT、SMALLINT、INT和BIGINT类型的取值范围相同。

浮点类型数据可以定义为REAL类型和FLOAT类型。其中，REAL类型占用4个字节的存储空间，可以精确到小数点后7位，其取值范围为-2.4E38~2.4E38；FLOAT类型占用8个字节的存储空间，可以精确到小数点后15位，其取值范围为-2.7E308~2.7E308。

2.4.2 数值类型

在Oracle数据库中，可以使用NUMBER(m,n)来定义数字类型的数据。其中，参数m表示存储数据的有效数字的位数；参数n表示小数点后的位数。例如NUMBER（5,2），第一个参数5表示显示的数字总位数为5，第二个参数表示小数点后的数字个数。如果把数据122.456插入到定义为NUMBER（5,2）的数据列中，则实际放入到该列的数据为122.46。

2.4.3 字符类型

在数据库中，字符类型是用来存储字符串值的。不同的数据库，字符类型的定义也不完全相同，下面以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，介绍在这3种数据库中字符类型不同的定义方法。

1. Oracle数据库

Oracle数据库中可以使用CHAR或者是VARCHAR2两种形式定义字符数据。其中，CHAR用来定义定长字符串，VARCHAR2用来定义可变长字符串。字符类型及其意义如表2.3所示。

表2.3 Oracle数据库中字符类型及其意义

字符类型	取值范围
CHAR(n)	定义定长的字符串（以字节为单位）1~字节
CHAR(n CHAR)	定义定长的字符串（以字符为单位）1~2000字节
VARCHAR2(n)	定义可变长的字符串（以字节为单位）1~4000字节
VARCHAR2(n CHAR)	定义可变长的字符串（以字符为单位）1~4000字节

- ❑ 如果不指定n的长度，默认值为1个字节的长度。
- ❑ 如果在定义为字符类型的列中存储汉字，则一个汉字占两个字节。



□ 对于VARCHAR2类型，如果实际存储的字符长度比参数n所设定的长度小，则Oracle数据库会将其长度自动调节到与该字符的实际长度相同。如果存储的字符串前后存在空格，则Oracle数据库会自动将空格删除。

□ 如果要存储大量的字符信息，建议使用CLOB类型。

例如，CHAR(200)表示定义的数据列中最多可以存放字符串的大小为200个字节。VARCHAR2(200 CHAR)表示定义的数据列中最多可以存放200个字符。

**注意** 使用CHAR定义的列存储的字符串所占空间是不可变的，使用VARCHAR2定义的列存储的字符串所占空间是可变的。

## 2. MySQL数据库

MySQL数据库中可以使用CHAR、VARCHAR、TEXT等形式定义字符数据。其中，CHAR用来定义定长字符数据，VARCHAR和TEXT用来定义可变长字符数据。字符类型及其意义如表2.4所示。

表2.4 MySQL数据库中字符类型及其意义

字符类型	取值范围
CHAR(n)	定义定长的字符串1~255个字符
VARCHAR(n)	定义可变长的字符串1~65535个字符
TINYTEXT	定义可变长的字符串1~65535个字符
TEXT	定义可变长的字符串1~2 <sup>16</sup> -1个字符
MEDIUMTEXT	定义可变长的字符串1~2 <sup>24</sup> -1个字符
LONGTEXT	定义可变长的字符串1~2 <sup>32</sup> -1个字符

□ CHAR(n)和VARCHAR(n)在定义字符类型时需要指定长度，而TINYTEXT、TEXT、MEDIUMTEXT和LONGTEXT在定义字符类型时不需要为其指定长度。

□ 如果在定义为字符类型的列中存储汉字，则一个汉字占两个字节。

## 3. Microsoft SQL Server数据库

Microsoft SQL Server数据库中，可以使用CHAR、VARCHAR、TEXT等形式定义字符数据。其中，CHAR用来定义定长字符数据，VARCHAR和TEXT用来定义可变长字符数据。如果字符数据超过了8KB，可以使用TEXT数据类型进行存储。

### 2.4.4 日期与时间类型

在数据库中，日期与时间类型是用来存储日期和时间值的。不同的数据库，日期与时间类型的定义也不完全相同，下面以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，介绍在这3种数据库中日期与时间类型不同的定义方法。

#### 1. Oracle数据库

在Oracle数据库中可以使用DATE、TIMESTAMP等形式定义日期时间类型的数据。其中DATE类型的数据在英文版本中日期默认的格式为DD-MON-YY的形式。例如10-SEP-09；在中文版本中的默认格式为日-月-年，如10-9月-09；TIMESTAMP类型在英文版中的默认的格式为YYYY-MM-DD HH.MM.



SS.AM如10-SEP-09 12.22.000000PM。在中文版本中的默认日期格式为‘日-月-年 时、分、秒’，例如，09-8月10 18.12.22.000000下午。它除了包含时、分、秒之外，还包含了秒的小数部分。秒值精确到小数点后6位。

## 2. MySQL数据库

在MySQL数据库中，可以使用DATE、TIME、DATETIME等形式定义日期时间类型。其中，DATE类型的数据默认格式为YYYY-MM-DD，例如，2009-08-10；TIME类型的数据默认格式为HH:MM:SS，例如，18:13:23；DATETIME类型的数据默认格式为YYYY-MM-DD HH:MM:SS，例如，2009-08-10 18:13:23。

## 3. Microsoft SQL Server数据库

在Microsoft SQL Server数据库中，可以使用DATETIME、SMALLDATETIME两种形式定义日期时间类型。其中，DATETIME类型需要8个字节的存储空间，其日期的取值范围从1753年1月1日到9999年12月31日，时间部分可以精确到2.33毫秒；SMALLDATETIME类型需要4个字节的存储空间，其日期的取值范围从1900年1月1日到2079年6月6日，时间部分可以精确到分钟。SMALLDATETIME类型的精确度没有DATETIME类型的精确度高。

在SQL Server 2008中，增加了四种DATETIME类型的数据，分别为DATE、TIME、DATETIMEOFFSET和DATETIME2。其中DATE类型只用来存储日期，其取值范围为0001-01-01~9999-12-31；TIME类型只用来存储时间，其取值范围为00:00:00.0000000~23:59:59.9999999；DATETIMEOFFSET类型保证了存储的日期和时间的时区一致性；DATETIME2是对DATETIME的扩展，其取值范围要比DATETIME的取值范围大，并可以通过参数设定小数的位数，最多可以精确到小数点后7位。

### 2.4.5 二进制类型

在数据库中，二进制类型是用来存储二进制数据的。不同的数据库，二进制类型的定义也不完全相同，在Oracle数据库中可以使用BLOB存储二进制数据信息，最多可以存储4GB。

在MySQL数据库中除了可以使用BLOB存储二进制的的数据以外，还可以使用TINYBLOB、MEDIUMBLOB、LONGBLOB等存储二进制类型的数据。在Microsoft SQL Server数据库中，可以使用BINARY、VARBINARY和IMAGE存储二进制数据信息。其中，在IMAGE数据类型中存储的数据是以位字符串存储的，它不是通过Microsoft SQL Server解释的，而是需要通过应用程序来解释。

## 2.5 SQL语句书写规范

在使用SQL语言执行数据的查询、更新等操作时，还需要了解SQL语言书写规范。这一节就来介绍SQL语言中一些主要的书写规范。

- SQL语言中不区分关键字的大小写。例如下面两种SQL语句的写法在数据库管理系统中都可以正确地执行。

```
SELECT teaID,teaName,age FROM T_teacher WHERE age > 30
select teaID,teaName,age from T_teacher where age > 30
```

其中，SELECT、FROM、WHERE是SQL语句中的关键字。一般情况下，书写SQL语言时，关键

## 零基础学SQL

字需要大写。本书在以后的SQL语言的讲解中，所有的关键字均以大写的形式给出。

- SQL语言中不区分列名和对象名的大小写。例如下面两种SQL语句的写法在数据库管理系统中都可以正确地执行。

```
SELECT teaID,teaName,age FROM T_teacher WHERE age > 30
SELECT TEAID,TEANAME,AGE FROM t_teacher WHERE AGE> 30
```

其中，teaID、teaName、age表示列名，T\_teacher表示表名。本书中对于列名和对象名使用SQL语句中的第一种写法。

- SQL语言中对数据库中的数据是大小写敏感的。
- SQL语言中单行注释可以使用“--”。使用“--”进行单行注释时，“--”后面至少要有一个空格。

```
SELECT teaID,teaName,age FROM T_teacher WHERE age > 30 -- 查询教师信息表中年龄大于30岁的教师信息
```

- 多行注释可以使用“/\*注释内容\*/”。其中，以“/\*”开头到“\*/”结尾的内容都属于被注释的内容。

```
/*
查询教师信息表中教师信息
要求查询的教师的年龄要大于30岁
*/
SELECT teaID,teaName,age FROM T_teacher WHERE age > 30
```

无论单行注释还是多行注释，都只是对SQL语句的解释说明，注释的内容并不会被执行。

- SQL语言中的语句可以写在一行，也可以写在多行上。如果要查询教师信息表中年龄大于30岁的教师信息，多行的SQL语句的写法如下：

```
SELECT teaID,teaName,age
FROM T_teacher
WHERE age > 30
```

## 2.6 小结

希望通过本章的学习，读者可以了解SQL语言的发展、SQL语句的分类以及SQL语言的特点。SQL语言中常用的数据类型包括整数类型、浮点类型、数值类型、日期时间类型、二进制类型等。在使用数据类型时，需要根据实际应用的需要来确定数据类型的长度、精度，以便选择适合的数据类型。

另外还需要掌握SQL语句的基本书写规范。在以后的章节中，将对SQL语句中数据定义语言（DDL）、数据查询语言（DQL）、数据操作语言（DML）、数据控制语言（DCL）以及事务控制语言的应用——进行介绍。

## 第二篇

# 数据库与数据表的创建和管理

### 第3章 数据库的创建与删除

在创建数据表之前，首先需要创建数据库。只有先创建数据库，才能在数据库中创建数据表。如果数据库不再需要，还可以将其删除。本章主要介绍使用SQL语句以及如何在MySQL 5.0 Command Line Client窗口和MySQL 5.0的用户图形界面下创建和删除方法。

本章重点：

- ☐ 使用SQL语句创建和删除数据库
- ☐ 在MySQL 5.0 Command Line Client窗口下创建和删除数据库
- ☐ 在MySQL 5.0的用户图形界面下创建和删除数据库

#### 3.1 创建数据库

使用SQL语句中的CREATE DATABASE可以创建一个数据库。另外，不同的数据库管理系统中也提供了创建数据库的方法。这一节就以MySQL 5.0数据库为例，介绍如何在MySQL 5.0数据库中创建一个数据库。

##### 3.1.1 使用SQL语句创建数据库

在创建数据表之前，首先需要创建数据库。只有先创建数据库，才能在数据库中创建数据表。创建数据库的语法规则如下：

```
CREATE DATABASE database_name
```

这里使用CREATE DATABASE语句创建一个数据库。其中，CREATE DATABASE是创建数据库的关键字；database\_name表示要创建的数据库的名字。在CREATE DATABASE语句后跟的就是数据库的名字。下面通过一个例子介绍一下如何使用CREATE DATABASE语句创建一个数据库。

下面，创建一个名为test\_STInfo的数据库。

```
CREATE DATABASE test_STInfo
```

这段SQL语句就是创建一个名为test\_STInfo数据库。CREATE DATABASE是创建数据库需要用到

## 零基础学SQL

的关键词，test\_STInfo是数据库的名字。

### 3.1.2 在MySQL 5.0 Command Line Client窗口下创建数据库

下面以MySQL 5.0数据库为例，看一下在MySQL 5.0数据库中，如何在MySQL 5.0 Command Line Client窗口下，使用CREATE DATABASE语句创建test\_STInfo数据库。

(1) 单击“开始”|“所有程序”|“MySQL 5.0”|“MySQL 5.0 Command Line Client”命令。运行后，在出现的MySQL 5.0 Command Line Client窗口中输入密码。其密码为root。

(2) 在MySQL 5.0 Command Line Client窗口中mysql>的后面使用CREATE DATABASE语句创建test\_STInfo数据库，并在该SQL语句的后面以分号结尾。

(3) 按“Enter”键，执行创建数据库的SQL语句，其执行后的结果如图3.1所示。

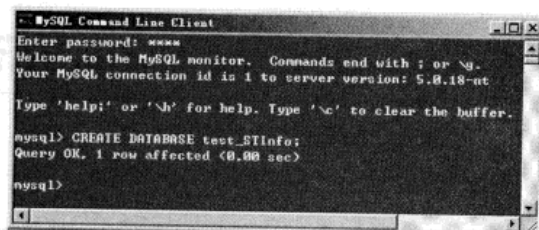


图3.1 创建test\_STInfo数据库

**注意** 在使用MySQL 5.0 Command Line Client窗口书写SQL语句时，在SQL语句的后面都需要加一个分号（；），否则SQL语句无法得到正确的执行。

在图3.1中，可以看到，使用CREATE DATABASE语句创建test\_STInfo数据库后，在其创建数据库的SQL语句下面，输出了如下的信息：

```
Query OK, 1 row affected (0.00 sec)
```

这段输出信息说明，在MySQL 5.0数据库中有一行数据发生了变化，表明CREATE DATABASE语句创建test\_STInfo数据库的SQL语句已经得到了正确地执行。

为了查看test\_STInfo数据库是否已经创建，可以使用鼠标双击“我的电脑”，进入到C:\Program Files\MySQL\MySQL Server 5.0\data的目录下，若可以看到在该目录下多出了一个test\_STInfo文件夹，则表明test\_STInfo数据库已经创建完成了。

**注意** 如果想在MySQL 5.0 Command Line Client窗口中使用test\_STInfo数据库，可以使用USE test\_STInfo语句。

### 3.1.3 在MySQL 5.0用户图形界面中创建数据库

当然，除了可以在MySQL 5.0 Command Line Client窗口中创建数据库之外，还可以在MySQL 5.0的MySQL 5.0用户图形界面中直接创建数据库。创建方法如下：

(1) 单击“开始”|“所有程序”|“MySQL”|“MySQL Query Browser”命令，会出现一个MySQL 5.0用户图形界面的确认界面。

(2) 在该界面的Server Host文本框中输入127.0.0.1；Port文本框中使用默认的端口号3306；在Username文本框中输入root，在Password文本框中输入密码root。然后在Default Schema文本框中输入一个数据库的名字。这里输入的是test\_STInfo。如果MySQL 5.0中没有该数据库，则系统会为其创建一个同名的数据库，如图3.2所示。

(3) 设置完成后，单击“OK”按钮进入MySQL 5.0的MySQL 5.0用户图形界面。在该MySQL 5.0用户图形界面的右侧Schemata选项下面对应的数据库中，会看到刚才创建的test\_STInfo数据库，如图3.3所示。

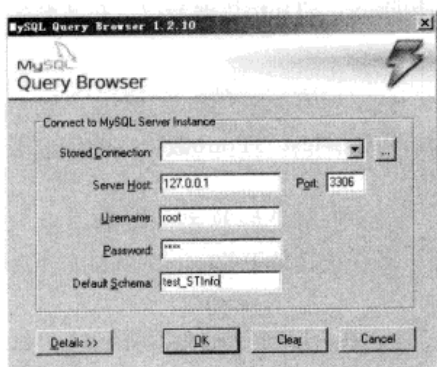


图3.2 MySQL 5.0用户图形界面的确认界面



图3.3 创建test\_STInfo数据库

## 3.2 删除数据库

使用SQL语句中的DROP DATABASE可以删除一个数据库。另外，不同的数据库管理系统中也提供了删除数据库的方法。这一节就以MySQL 5.0数据库为例，介绍如何在MySQL 5.0数据库中删除一个数据库。

### 3.2.1 使用SQL语句删除数据库

如果想删除数据库，可以使用DROP DATABASE语句将其删除。其语法规则如下：

```
DROP DATABASE database_name
```

其中，DROP DATABASE表示用于删除数据库的关键字，database\_name用来表示要删除的数据库的名字。例如，如果要删除test\_STInfo数据库，则可以使用如下的SQL语句。

```
DROP DATABASE test_STInfo
```

这段SQL语句就是删除一个名为test\_STInfo的数据库。DROP DATABASE是删除数据库需要用到的关键字，test\_STInfo是数据库的名字。

### 3.2.2 在MySQL 5.0 Command Line Client窗口下删除数据库

下面以MySQL 5.0数据库为例，看一下在MySQL 5.0数据库中，如何在MySQL 5.0 Command Line Client窗口下，使用DROP DATABASE语句删除test\_STInfo数据库。

- (1) 单击“开始”|“所有程序”|“MySQL 5.0”|“MySQL 5.0 Command Line Client”命令。运行后，在出现的MySQL 5.0 Command Line Client窗口中输入密码。其密码为root。
- (2) 在MySQL 5.0 Command Line Client窗口中mysql>的后面使用DROP DATABASE语句将test\_STInfo数据库删除，并在该SQL语句的后面以分号结尾。
- (3) 按“Enter”键，执行删除数据库的SQL语句，其执行后的结果如图3.4所示。



零基础学SQL

**注意** 在使用MySQL 5.0 Command Line Client窗口书写SQL语句时，在SQL语句的后面都需要加一个分号（;），否则SQL语句无法得到正确的执行。

在图3.4中，可以看到，使用DROP DATABASE语句删除test\_STInfo数据库后，在其删除数据库的SQL语句下面，输出了如下的信息：

```
Query OK, 0 rows affected (0.16 sec)
```

在这段输出信息中Query OK表明DROP DATABASE语句删除test\_STInfo数据库的SQL语句已经得到了正确的执行。

为了查看test\_STInfo数据库是否已经成功删除，可以使用鼠标双击“我的电脑”，进入到C:\Program Files\MySQL\MySQL Server 5.0\data的目录下，若在该目录下的test\_STInfo文件夹已经不见了，则表明test\_STInfo数据库已经被删除了。

也可以在MySQL 5.0 Command Line Client窗口中验证，在3.1.2小节最后提到过，如果想使用test\_STInfo数据库，可以使用USE test\_STInfo语句。由于已经使用DROP DATABASE语句将test\_STInfo数据库删除了，如果在MySQL 5.0 Command Line Client窗口输入USE test\_STInfo语句，应该得到一个错误信息。下面就来看一下在MySQL 5.0 Command Line Client窗口输入USE test\_STInfo语句后的显示结果，如图3.5所示。

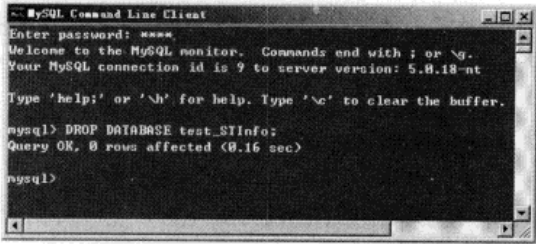


图3.4 删除test\_STInfo数据库

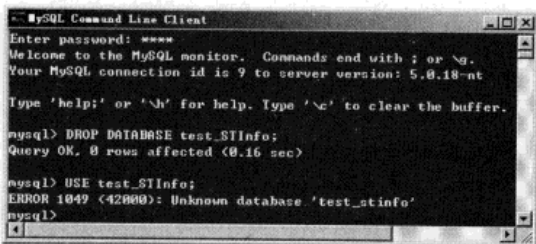


图3.5 删除test\_STInfo数据库后使用USE test\_STInfo语句

从图3.5中可以看到，删除test\_STInfo数据库后在MySQL 5.0 Command Line Client窗口使用USE test\_STInfo语句，会得到一个错误信息，表示数据库管理系统没有找到test\_STInfo数据库，说明该数据库已经被成功删除了。

### 3.2.3 在MySQL 5.0用户图形界面中删除数据库

当然，除了可以在MySQL 5.0 Command Line Client窗口中删除数据库之外，还可以在MySQL 5.0的MySQL 5.0用户图形界面中直接删除数据库。删除方法如下：

- (1) 单击“开始”|“所有程序”|“MySQL”|“MySQL Query Browser”命令，会出现一个MySQL 5.0用户图形界面的确认界面。
- (2) 在Username文本框中输入root，在Password文本框中输入密码root。在Default Schema文本框中会显示刚才创建的test\_STInfo数据库的名字。
- (3) 设置完成后，单击“OK”按钮进入MySQL 5.0的MySQL 5.0用户图形界面。在该MySQL 5.0用户图形界面的右侧Schemata选项下对应的数据库中，会看到刚才创建的test\_STInfo数据库。
- (4) 右击test\_STInfo数据库，在出现的列表框中，选择“Drop Schema”命令，如图3.6所示。



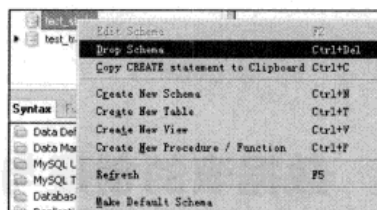


图3.6 在MySQL 5.0用户图形界面删除数据库

(5) 在出现的确认删除的对话框中，单击“OK”按钮，完成test\_STInfo数据库的删除。

### 3.3 小结

本章主要介绍了如何使用SQL语句创建和删除一个数据库。只有先创建数据库，才能在数据库中创建数据表。因此在学习创建数据表以及如何使用SELECT语句查询数据之前，有必要了解数据库的创建过程。如果数据库不再需要，还可以将其删除。

## 第4章 数据表的创建与更新

数据表是关系数据库中操作的级别对象。在关系数据库中，通过数据表来存储数据记录。在数据表中会经常用到主键、外键、约束和索引。本章将对数据表中涉及的数据类型以及主键、外键、约束和索引等概念和使用方法做一个全面的介绍。另外还将介绍数据表的创建方法，以及如何对表中的记录进行增加和修改操作、如何删除数据表等内容。

本章重点：

- ☐ 主键、外键、约束、索引的概念
- ☐ 数据表的创建
- ☐ 约束的使用
- ☐ 索引的使用与优化
- ☐ 更新数据表
- ☐ 删除数据表
- ☐ 数据库test\_STInfo中数据表的介绍

### 4.1 数据库中的表

在关系数据库中，数据表是存储数据的基本单元，由行和列两部分组成。它是根据数据库设计阶段的E-R图转换而来。在数据表中除了行、列、数据记录、属性、字段等基本概念外，还会涉及主键、外键、索引、约束等概念。这些概念在数据表的设计和创建的过程中会经常用到，本节就对这些概念做一个简单的介绍。

#### 4.1.1 数据记录、属性、字段、列和行

在关系数据库中，数据表是存储数据的基本单元，由行和列两部分组成。其中，行用来描述实体中的具体数据，在数据表中每一行中的数据被称为一条数据记录，也可以简称为记录；字段是表中的一列，用来保存数据表中某一条记录中的特定信息。字段在关系数据库中也可以称为列。

例如，在表4.1中，字段stuID用来表示学生编号，字段stuName用来表示学生的姓名，字段age用来表示学生的年龄。每一列包含了一个特定字段的全部信息，例如数据表中基于字段stuName的列就包含了学生姓名的信息，这一列中的学生姓名信息包括赵亮、王海、张明、郑茹；学生信息表中的每一行都描述了一个学生的基本信息，例如数据表中第一行对应的数据s102203、赵亮、23就是一条数据记录，在这条记录中包含有学生编号、学生姓名以及学生

表4.1 学生信息表

stuID	stuName	age
s102203	赵亮	23
s115263	王海	23
s206363	张明	22
s112303	郑茹	21

年龄等几个字段，它们都是用来表示学生编号为s102203学生的基本信息的。

4.1.2 主键

为了保证在一张数据表中不会出现两个完全相同的数据记录，需要为每一张数据表都定义一个主键。主键作为数据表中的唯一标示，保证了一条记录的唯一性。通过主键可以区分数据表中的每一条记录。在创建数据表时，需要保证被定义为主键的列的列值唯一，并且不能为空值（即NULL值）。

主键在关系模型中用来约束实体完整性。所谓实体完整性约束，是指对数据表中行的完整性约束。在实体完整性中，需要有一个主键来唯一标示数据表中每一行的数据记录，数据表中的主键唯一且不能为空值。

例如，在表4.1中，学生编号字段stuID就是这张表的主键，每一个主键对应其中一名学生，在学生信息表中主键对应的值不能为空，更不会重复出现，也就是说不能出现两个stuID是s102203的学生信息。

当然，主键列并不一定就只有一列，也可以将多个列组合在一起作为表的主键。例如在表4.2中，有3个列，列stuID用来表示学生编号，列curID用来表示课程编号，列result用来表示学生成绩。从这几条记录中可以看到，其中的任何一列单独拿出来，都不可能作为主键唯一标示一条学生的成绩信息。如果想要唯一地标示一条学生的记录信息，需要将列stuID和列curID共同作为学生成绩表的主键，这样才能唯一地标示一条学生的成绩信息。

表4.2 学生成绩表

stuID	curID	result
s102203	t105	85
s102203	t232	75
s102203	t333	60
s102203	t321	90

**注意** 主键可以定义在多个列上，并不一定只定义在一个列上。也就是说，在定义数据表的时候，可以将数据表中的多个列合在一起作为该表的主键。

4.1.3 外键

外键是用来定义表与表之间的关系的。在数据表中，外键是这样定义的：如果属性列F是关系B中一个属性（并不是关系B的主键），并且属性列F是关系A中的主键，则F就是关系B的外键。关系A中的表被称为主表，关系B中的表被称为主表的从表。

外键在关系模型中用来约束参照完整性。所谓参照完整性约束，是指表与表之间的约束。在参照完整性中，从表中的每一条数据记录中的外键值都必须存在于主表中。对于建立了关联关系的两个数据表来说，对其中一个数据表的增加、修改或者删除数据的操作都会对另一个数据表中的记录产生影响。

例如，在表4.1和表4.2中，列stuID和列curID两个列联合作为该表的主键。则列stuID就是学生成绩表的外键。其中，表4.1是主表，表4.2是主表的从表。例如，如果想向学生成绩表中插入一条学生成绩记录，数据库管理系统首先会对学生信息表进行检查，看这个学生是否存在于学生信息表中，如果存在，才会允许执行插入操作；如果该名学生在学生信息表中不存在，则执行插入操作时，数据库管理系统会报错，拒绝执行插入操作。

同样，定义了外键约束的两个表在数据修改和数据删除操作时也会存在一些限制。有关定义了外键约束的两个表之间进行增加、修改和删除数据的操作，可以参看12.1.2、13.1.2和14.1.2节中介绍的内容。

#### 4.1.4 索引

在实际应用中，为了加快访问速度，节省访问时间，一般都需要使用索引进行查询。例如，在使用电话簿查询电话时，为了减少查询的时间，一般都会使用电话簿中提供的企业名称作为索引来查询；在使用一本书学习某一部分知识的时候，一般都需要翻看书后提供的索引，一般书后的索引都会以字母顺序将相关的主题信息列出，通过这个索引的指引，读者可以很快查找到想要的信息，而不需要为了查询某一个知识点而将书中所有的内容都翻看一遍，节省了查阅的时间，也保证了学习效率。

数据库中的索引与书后提供的索引的功能相同，在数据库的应用中，往往一张数据表中会包含上千条甚至上万条记录，因此为了加快对数据表的访问，通常需要在数据表中建立适当的索引。通过建立索引，在查询数据表中的数据时，数据库可以很快地将其找到，而不用扫描整个数据表。

索引是一个指向数据表中数据的指针，指向索引字段在数据表中的物理位置。如果在执行查询操作时，WHERE子句中指定的字段是被设置为索引的字段，则数据库会首先在索引中对指定的值进行查询，并返回查询的数据在数据表中的位置。如果在执行查询操作时，WHERE子句中指定的字段没有设置为索引的字段，那么数据库会对查询数据表中的每一行数据记录进行扫描。因此适当地创建索引，可以加快数据的检索速度，提高对数据的访问效率，提供数据查询的性能。

当然，索引本身也有一些弊端，例如，索引会占用大量的硬盘空间；随着数据列的增加，创建和维护索引的时间也会随之增加；在对数据进行增加、删除和修改等更新操作的时候，需要对索引进行维护，降低更新数据的速度。因此，对那些不是在查询过程中经常用到的列以及在数据表中经常需要进行增加、删除和修改等更新操作的列就不适合建立索引。

虽然创建索引可以提高查询的速度，但是由于索引本身会占用物理空间以及维护索引可能带来的时间的损耗，所以在为数据表中的列创建索引时，并不是为数据表中的每一个列都要创建索引，那样做反而不会起到提高查询效率的作用。因此需要在数据表的适当的列上创建索引。一般可以在下面这些列中创建索引。

- ☐ 在主键列中创建索引。
- ☐ 多表连接时，在经常使用的连接列上创建索引。
- ☐ 在经常使用WHERE子句查询的列上创建索引。
- ☐ 在经常进行分组（GROUP BY）和排序（ORDER BY）的列上创建索引。

#### 4.1.5 约束

为了保证数据的完整性，需要使用数据库约束。完整性约束包括对表的约束和对列的约束。表约束主要包括唯一约束、主键约束、外键约束和检查约束，列约束除了包括唯一约束、参照约束和检查约束之外，还有非空约束。

- ☐ 唯一约束（UNIQUE）：保证使用唯一约束的某一列或者一组列中没有相同的值，即保证列的值的唯一性。但是唯一约束中可以允许在列中插入空值（即NULL值）。
- ☐ 主键约束（PRIMARY KEY）：保证使用主键约束的列中只能有唯一的值，并且不能包含空值。数据表中每一列只能定义一个PRIMARY KEY。
- ☐ 外键约束（FOREIGN KEY）：保证表的参照完整性。确保对一个表的数据操作不会对与之关联的表造成不利的影响。
- ☐ 检查约束（CHECK）：限制列的取值范围或者取值条件。可以为一个列定义多个CHECK约束。

□ 非空约束（NOT NULL）：只用来约束列。在向该列插入数据时不允许插入空值。

## 4.2 创建数据表

在创建完数据库之后，就可以在数据库中创建数据表了。创建数据表可以通过使用CREATE TABLE语句。使用CREATE TABLE语句创建数据表的语法格式如下：

```
CREATE TABLE table_name(  
    column_name1 datatype1 [constraint_condition1]  
    [, column_name2 datatype2 [constraint_condition2]]...  
)
```

这段使用CREATE TABLE创建数据表。其中，CREATE TABLE是创建数据表的关键字；table\_name表示要创建的数据表的名字；column\_name1用来指定数据表的列名；datatype1指定列名为column\_name1中的数据类型；constraint\_condition1指定列名为column\_name1中的完整性约束条件；column\_name2、datatype2和constraint\_condition2与column\_name1、datatype1和constraint\_condition1表示的意义相同。

这里所说的完整性约束条件可以是唯一约束（UNIQUE）、主键约束（PRIMARY KEY）、外键约束（FOREIGN KEY）、检查约束（CHECK）以及非空约束（NOT NULL）。有关这些约束的详细的使用方法可以参考4.3节。

**注意** 完整性约束条件可以定义在列级上，也可以定义在表级上。列级完整性约束是指在定义列和指定列中数据类型之后就定义完整性约束条件；表级的完整性约束是指在定义了所有列之后再定义完整性约束条件。

在创建数据表的CREATE TABLE语句中，可以为数据表创建多个列，数据表中的每个列都需要指定列名和该列对应的数据类型，列名对应的完整性约束条件是可选的。下面通过一个例子介绍一下如何使用CREATE TABLE语句创建一个数据表。

**例4.1** 创建表名为T\_student的学生信息表。

```
CREATE TABLE T_student (  
    stuID VARCHAR (15) PRIMARY KEY,  
    stuName VARCHAR (10) NOT NULL,  
    age INT NOT NULL,  
    sex VARCHAR (2) NOT NULL,  
    birth DATETIME NOT NULL  
)
```

这段SQL语句是使用CREATE TABLE创建一张学生信息表。其中，T\_student表示创建数据表的表名，在T\_student表中，共指定了5个列用来描述学生信息。

□ stuID指定数据表的列名，列stuID用来表示学生编号；VARCHAR (15)用来指定stuID列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为15；PRIMARY KEY用来指定列stuID的完整性约束条件，这里将列stuID设为主键，表示列stuID的值唯一并且不能为空值（NULL值）。

□ stuName指定数据表的列名，列stuName用来表示学生姓名；VARCHAR (10)用来指定stuName列



的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为10；NOT NULL用来指定列stuName的完整性约束条件，这里将列stuName设定为非空，表示该列中不允许存在空值（NULL值）。

❑ age指定数据表的列名，列age用来表示学生年龄；INT用来指定age列的数据类型，这里将其指定为INT类型，表示学生的年龄存储在数据库中都应该为整数；NOT NULL用来指定列age的完整性约束条件，这里将列age设定为非空，表示该列中不允许存在空值（NULL值）。

❑ sex指定数据表的列名，列sex用来表示学生性别；VARCHAR (2)用来指定sex列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为2；NOT NULL用来指定列sex的完整性约束条件，这里将列sex设定为非空，表示该列中不允许存在空值（NULL值）。

❑ birth指定数据表的列名，列birth用来表示学生出生日期；DATETIME用来指定birth列的数据类型，这里将其指定为DATETIME类型，即将学生的出生日期设定为日期时间类型；NOT NULL用来指定列birth的完整性约束条件，这里将列birth设定为非空，表示该列中不允许存在空值（NULL值）。

**注意** 使用CREATE TABLE语句创建数据表后，数据表中不会存在任何记录，如果想向数据表中插入记录，需要使用INSERT INTO语句。有关使用INSERT INTO语句向数据表中插入数据记录的方法可以参看第12章。

下面以MySQL 5.0数据库为例，看一下在MySQL 5.0数据库中，如何在MySQL 5.0 Command Line Client窗口下，使用CREATE TABLE语句创建学生信息表（T\_student）。这里将学生信息表（T\_student）再创建到test\_STInfo数据库中。

(1) 进入MySQL 5.0 Command Line Client窗口中输入密码。其密码为root。

(2) 在MySQL 5.0 Command Line Client窗口中mysql>的后面使用USE test\_STInfo，告诉数据库管理系统，现在要使用的是test\_STInfo数据库。

(3) 使用CREATE TABLE语句创建学生信息表（T\_student），并在CREATE TABLE语句的后面以分号结尾。

(4) 按“Enter”键，执行创建数据表的SQL语句，其执行后的结果如图4.1所示。

为了查看学生信息表（T\_student）是否已经创建，可以使用鼠标双击“我的电脑”，进入到C:\Program Files\MySQL\MySQL Server 5.0\data\test\_stinfo的目录下，若可以看到在该目录下多出了一个t\_student.frm文件，则表明学生信息表T\_student已经创建完成了。

当然，除了可以在MySQL 5.0 Command Line Client窗口中创建数据库之外，还可以在MySQL 5.0的MySQL 5.0用户图形界面中直接创建数据表。

(1) 单击“开始”|“所有程序”|“MySQL”|“MySQL Query Brower”命令，输入用户名、密码后，在Default Schema选项对应的文本框中输入一个数据库的名字。这里输入的是test\_STInfo。

(2) 进入到MySQL 5.0的图形用户界面中，在该界面上方的空白处写入创建学生信息表（T\_student）

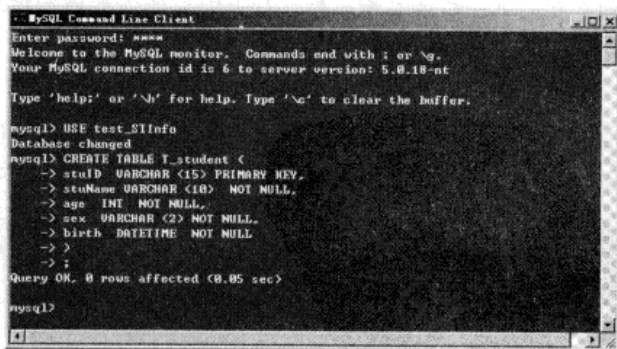


图4.1 创建学生信息表（T\_student）



的SQL语句。

(3) 单击空白处右侧的“Execute”按钮，如果执行的SQL语句没有错误，则在该界面的下方会出现如下提示信息：

Query returned no resulted

这段信息表示数据表已经得到了正确的创建。

(4) 选中MySQL 5.0用户图形界面的右侧Schemata选项的下面对应的名为test\_STInfo数据库，右击test\_STInfo数据库，选择“Refresh”命令（或者按F5键，刷新test\_stinfo数据库，如图4.2所示。

(5) 刷新操作执行后，在test\_stinfo数据库下方就可以看到一个名为T\_student的数据表，如图4.3所示。

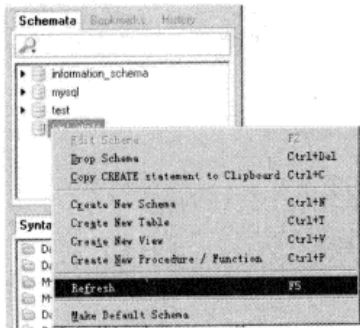


图4.2 刷新test\_STInfo数据库



图4.3 在MySQL 5.0用户图形界面中显示学生信息表（T\_student）

例4.1给出的是创建数据表的一个比较简单的例子，在实际应用中，有时需要创建的数据表可能会很复杂，例如，创建的数据表做主键的列可能不只一列，而是多个列组合起来作为数据表的主键；两个表在相互关联时，需要为数据表定义外键约束；为数据表创建适当的索引等。这些在数据表的创建中也是经常会用到的，因此在创建数据表时有必要了解约束和索引是如何使用的。有关在创建数据表中使用约束和索引的内容将在4.3节和4.4节中做详细的介绍。

**说明** 在CREATE TABLE语句中，还可以使用子查询创建数据表。有关使用子查询创建数据表的方法可以参看9.7节。

### 4.3 使用约束

为了保证数据的完整性，需要使用数据库约束。约束主要包括唯一约束（UNIQUE）、主键约束（PRIMARY KEY）、外键约束（FOREIGN KEY）、检查约束（CHECK）和非空约束（NOT NULL）。这一节将介绍这几种约束的作用以及它们在创建数据表的过程中是如何定义的。

#### 4.3.1 唯一约束

唯一约束（UNIQUE）用来保证某一列或者一组列中没有相同的值。如果为列定义了唯一约束，则该列中不允许出现重复的值，但是允许列中存在空值（即NULL值）。唯一约束既可以定义的表级上，也可以定义在列级上。一般在为列创建唯一约束后，数据库会自动为该列建立一个唯一索引，其索引

名与约束名是相同的。

**例4.2** 创建院系信息表，并为院校编号所在列定义唯一约束。

```
CREATE TABLE T_dept (  
    deptID VARCHAR (15) UNIQUE,  
    deptName VARCHAR (10)
```

在这段SQL语句中，使用CREATE TABLE语句创建院系信息表，并为列deptID定义唯一约束。其中，T\_dept是表的名字，这里T\_dept表示院系信息表。在T\_dept表中，共指定了2个列用来描述院系信息。

□ deptID指定数据表的列名，列deptID表示院系编号，VARCHAR (15)用来指定deptID列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为15，UNIQUE用来指定列deptID的完整性约束条件，这里将其定义为唯一约束。

□ deptName表示院系名称，VARCHAR (10)用来指定deptName列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为10。

选中MySQL 5.0用户图形界面的右侧Schemata选项下的test\_STInfo数据库中的院系信息表T\_dept，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，在出现的表编辑对话框中，选择对话框下方的“Indices”选项卡，可以看到为院系信息表T\_dept中列deptID定义的唯一约束信息，如图4.4所示。

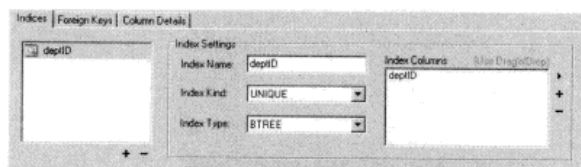


图4.4 唯一约束信息

在图4.4中，“Indices”选项卡下描述了对院系信息表T\_dept中列deptID定义的唯一约束信息。在“Index Settings”选项卡下指定了数据表T\_dept中唯一约束的信息。其中，“Index Name”指定了索引的名字，“Index Kind”指定了索引的类型为唯一索引，“Index Type”指定了索引类型；在界面的右下方空白处，在Index Columns选项卡下指定了唯一索引的列名为deptID。

**注意** 在为列创建唯一约束后，数据库会为该列建立一个唯一索引，其索引名与约束名是相同的。

### 4.3.2 主键约束

主键约束 (PRIMARY KEY) 是用来保证使用主键约束的某一列或者一组列中有唯一的值，并且不能包含空值 (即NULL值)。数据表中每一列只能定义一个PRIMARY KEY。一般在为列创建主键约束后，数据库会自动为该列建立一个主索引，其索引名与约束名是相同的。

在创建数据表时，如果希望将多个列组合起来作为一个数据表中的主键，可以在PRIMARY KEY关键字后使用括号，将需要定义为主键的列放到PRIMARY KEY关键字后面的括号中。括号中的多个列之间需要使用逗号分隔。

**例4.3** 创建成绩信息表。

```
CREATE TABLE T_result(  
    stuID VARCHAR (15) ,  
    curID VARCHAR (15) ,  
    result DOUBLE ,  
    PRIMARY KEY (stuID,curID)  
)
```

在这段SQL语句中，使用CREATE TABLE语句成绩信息表，并将列stuID和列curID联合作为主键。其中，T\_result是表的名字，这里T\_result表示成绩信息表。在T\_result表中，共指定了3个列用来描述学生的成绩信息。

- ❑ stuID指定数据表的列名，列stuID表示学生编号，VARCHAR (15)用来指定stuID列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为15。
- ❑ curID指定数据表的列名，列curID表示院系编号，VARCHAR (15)用来指定curID列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为15。
- ❑ 列result表示学生成绩，DOUBLE用来指定result列的数据类型，这里将其定义为DOUBLE类型的数据。
- ❑ PRIMARY KEY表示主键约束，这里将列stuID和列curID联合作为成绩信息表的主键。

选中MySQL 5.0用户图形界面的右侧Schemata选项下test\_stinfo数据库中的成绩信息表T\_result，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，在出现的表编辑对话框中，选择对话框下方的“Indices”选项卡，可以看到为成绩信息表T\_result中列stuID和列curID定义的主键约束信息，如图4.5所示。

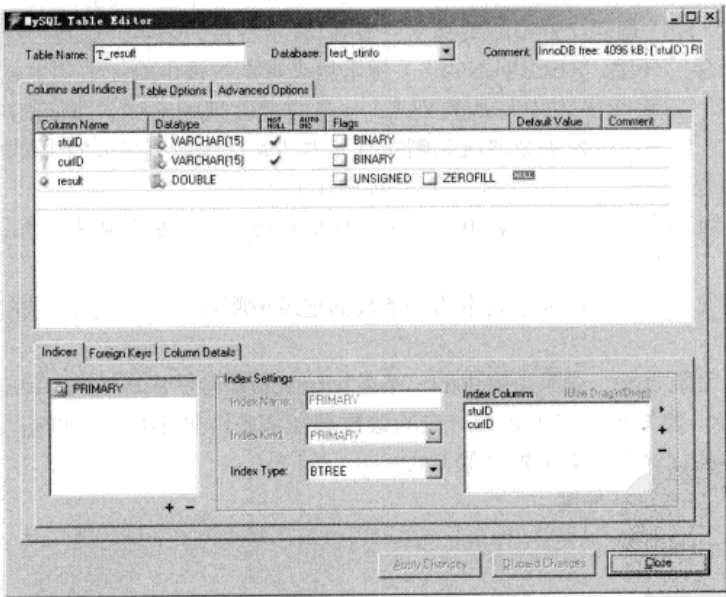




图4.5 主键约束信息

在图4.5中，在表编辑对话框的上方，从“Columns and Indices”选项卡下的表格中可以看到列stuID和列curID前面都有一个的图标，图标表示列stuID和列curID为成绩信息表T\_result的主键列。

“Indices”选项卡下方描述了对成绩信息表T\_result中列stuID和列curID定义的主键约束信息。其中，在表编辑对话框的左下方的空白处有一个PRIMARY关键字，在界面的右下方空白处，在Index Columns选项下指定了PRIMARY关键字对应的主键为列stuID和列curID。

**注意** 在为列创建主键约束后，数据库会为该列建立一个主索引，其索引名与约束名是相同的。

### 4.3.3 外键约束

首先来考虑这样一个问题，现在由于某种原因需要将一名学生在学校的全部信息删除。这里涉及学生信息的一共有两个表，一个是学生信息表，一个是成绩信息表。学生信息表保存的是学生的基本信息，成绩信息表保存的是学生选课的课程成绩信息。如果没有定义外键约束，将该名学生从学生信息表中删除时，而没有将该名学生的选课的课程成绩信息删除，这样就会造成两个关联表之间数据的不一致。

同样的问题也可能出现在对两个关联表中数据的修改操作中。如果是多个表之间存在关联关系的话，不定义外键约束，类似上面的情况很可能会发生。为了避免由于操作不当而引起的关联表中数据的不一致，有必要为关联表之间定义外键约束。

外键约束（FOREIGN KEY）主要是用来定义两个表之间的关系。外键约束保证了表的参照完整性，确保对一个表的数据操作不会对与之关联的表造成不利的影响。定义外键的语法格式如下：

```
FOREIGN KEY[表名1](列名1) REFERENCES 表名2(列名2)
[ON UPDATE [CASCADE]|[SET NULL]|[RESTRICT]]
[ON DELETE[CASCADE]|[SET NULL]|[RESTRICT]]
```

其中，FOREIGN KEY是定义外键的关键字；表名1表示从表的名字，它是可选的；列名1指定数据表中用于外键约束条件的外键；REFERENCES关键字用来指定主表中的表名和主表中的关键列；表名2表示主表的名字；列名2表示主表中与从表列名1对应的主键列的名字；其后的ON UPDATE和ON DELETE分别指明了在对表中的数据做修改和删除时，主从表之间所要采取的主要的操作方式，它们是可选的。下面以删除操作为例，分别讲解一下这3种操作方式。

- ☐ CASCADE：级联删除。如果主表中的一条数据记录被删除，那么从表中与之相对应的数据也将被一起删除。
- ☐ SET NULL：置空删除。如果主表中的一条数据记录被删除，那么从表中与之相对应的数据也将被设置为空值。
- ☐ RESTRICT：受限删除。如果主表中的一条数据记录被删除，则在执行DELETE命令时数据库管理系统会报错，通知用户与主表相对应的该数据在从表中仍然存在，但是与主表相对应的该数据在从表中不会被删除。它是默认的方式。

例4.4 为成绩信息表定义外键。

```
CREATE TABLE T_result(
stuID VARCHAR (15) ,
curID VARCHAR (15) ,
result DOUBLE ,
FOREIGN KEY(stuID) REFERENCES T_student(stuID) ON DELETE CASCADE,
PRIMARY KEY (stuID,curID)
)
```

这段SQL语句在例4.3的基础上，增加了一个创建外键的语句FOREIGN KEY(stuID) REFERENCES T\_student(stuID)。其中，FOREIGN KEY表示创建外键约束的关键字；stuID表示为成绩信息表（T\_result）中的表示学生编号的列stuID定义外键约束；REFERENCES T\_student(stuID)表示将列stuID定义为一个指向学生信息表T\_student的主键stuID中的外键，并使用ON DELETE CASCADE定义了其删除方式为级联删除。

选中MySQL 5.0用户图形界面的右侧Schemata选项下的test\_STInfo数据库中的成绩信息表T\_result，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，在出现的表编辑对话框中，选择对话框下方的“Foreign Keys”选项卡，可以看到为成绩信息表T\_result中列stuID定义的外键约束信息，如图4.6所示。

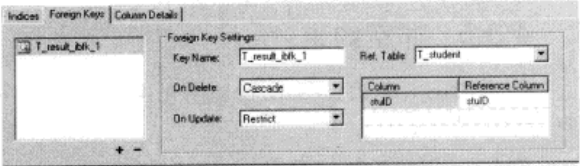


图4.6 外键约束

在图4.6中，“Foreign Key Settings”选项下描述了对数据表T\_result中列stuID定义的外键约束信息。其中，“Key Name”表示主键的名字；“Ref.Table”指定了表T\_result对应的主表，这里表T\_result对应的主表为T\_student（学生信息表）；On Delete指定了数据删除方式，这里的删除方式指定为Cascade（级联删除）；On Update指定了数据修改方式，这里的修改方式默认指定为Restrict（受限修改）。

**注意** 在定义外键约束时，其默认的修改和删除方式都是RESTRICT。由于例4.4的SQL语句指定了列stuID的删除方式为CASCADE（级联删除），而没有指定列stuID的修改方式，因此，数据库中就使用其默认的修改方式RESTRICT。

下面请读者思考一下，如果想在列curID中也定义一个指向课程信息表T\_curriculum的主键curID中的外键约束，并将其删除方式定义为级联删除，那么上述的SQL语句应该如何完成呢？

### 4.3.4 检查约束

检查约束（CHECK）是用来限制列的取值范围或者取值条件，使用CHECK约束可以保证数据规则的一致性。可以为一个列定义多个CHECK约束，当为列定义了CHECK约束后，该列中所对应的数据必须满足指定的约束条件。

**例4.5** 定义课程信息表。

```
CREATE TABLE T_curriculum(  
    curID VARCHAR (15) PRIMARY KEY,  
    curName VARCHAR (10),  
    credit INT,  
    CHECK(credit BETWEEN 3 AND 8)  
)
```

在这段SQL语句中，使用CREATE TABLE语句创建课程信息表，并使用CHECK关键字对列credit的取值范围进行约束。其中，T\_curriculum是表的名字，这里T\_curriculum表示课程信息表。在T\_curriculum表中，共指定了3个列用来描述课程信息。

- ❑ curID指定数据表的列名，列curID表示院系编号，VARCHAR (15)用来指定curID列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为15。并使用PRIMARY KEY关键字将该列定义为主键列。
- ❑ curName指定数据表的列名，列curName表示课程的名字，VARCHAR (10)用来指定curName列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为10。
- ❑ 列credit表示课程学分，INT用来指定credit列的数据类型，这里将课程学分所在的列定义为整型类型的数据。



- ❑ 使用CHECK关键字对表示课程学分的列credit的取值范围进行约束。这里将课程学分的取值范围设置在3到8之间。如果插入的数据小于3或者大于8，则SQL语句执行时会显示错误的信息。

#### 4.3.5 非空约束

非空约束（NOT NULL）是用来保证在向该列插入数据时不允许插入空值（即NULL值）。非空约束只能用来约束列。

例4.6 创建教师信息表。

```
CREATE TABLE T_teacher (  
teaID VARCHAR (15) PRIMARY KEY,  
teaName VARCHAR (10) NOT NULL,  
age INT NOT NULL,  
sex VARCHAR (2) NOT NULL,  
deptID VARCHAR (15),  
dept VARCHAR (20) NOT NULL,  
profession VARCHAR (10)  
)
```

这段SQL语句是使用CREATE TABLE创建一张教师信息表。其中，T\_teacher表示创建数据表的表名，在T\_teacher表中，共指定了7个列用来描述教师信息。

- ❑ teaID指定数据表的列名，列teaID用来表示教师编号；VARCHAR (15)用来指定teaID列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为15；PRIMARY KEY用来指定列teaID的完整性约束条件，这里将列teaID设为主键，表示列teaID的值唯一并且不能为空值（NULL值）。
- ❑ teaName指定数据表的列名，列teaName用来表示教师姓名；VARCHAR (10)用来指定teaName列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为10；NOT NULL用来指定列teaName的完整性约束条件，这里将列teaName设定为非空，表示该列中不允许存在空值（NULL值）。
- ❑ age指定数据表的列名，列age用来表示教师年龄；INT用来指定age列的数据类型，这里将其指定为INT类型，表示教师的年龄存储在数据库中都应该是整数；NOT NULL用来指定列age的完整性约束条件，这里将列age设定为非空，表示该列中不允许存在空值（NULL值）。
- ❑ sex指定数据表的列名，列sex用来表示教师性别；VARCHAR (2)用来指定sex列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为2；NOT NULL用来指定列sex的完整性约束条件，这里将列sex设定为非空，表示该列中不允许存在空值（NULL值）。
- ❑ deptID指定数据表的列名，列deptID用来表示教师所在的院系编号；VARCHAR (15)用来指定deptID列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为15。
- ❑ dept指定数据表的列名，列dept用来表示教师所在的院系；VARCHAR (20)用来指定dept列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为20；NOT NULL用来指定列dept的完整性约束条件，这里将列dept设定为非空，表示该列中不允许存在空值（NULL值）。
- ❑ profession指定数据表的列名，列profession用来表示教师的职称；VARCHAR (10)用来指定profession列的数据类型，这里将其指定为VARCHAR类型，并且设定该列中字符串长度为10。



在创建的教师信息表（T\_teacher）中，表示教师姓名的列teaName、表示教师年龄的列age、表示教师性别的列sex和表示教师所在院系的列dept都定义为NOT NULL类型的，也就是这几个列在数据插入时不允许插入空值。

## 4.4 使用索引

在实际应用中，为了加快查询的效率，都会为数据表创建适当的索引。索引就是一个指向数据表中数据的指针。通过为数据表建立适当的索引可以提高SQL语句对数据表的访问速度。这一节就来介绍如何在数据库中使用SQL语句创建和删除索引，有关优化索引的内容可以参看26.1节。

### 4.4.1 索引的分类

索引主要可以分为唯一索引、主索引、单列索引、复合索引以及聚簇索引等，下面分别来介绍一下这几种索引。

- ❑ 唯一索引：在数据表中使用UNIQUE关键字可以为一个数据列定义一个唯一索引。唯一索引中每一个索引值只对应数据表中的一条记录，它保证了数据列中记录的唯一性。如果在向数据表中插入一条记录时，数据库会对要插入的这条记录进行检查，如果发现该条记录中的值在定义了唯一索引的列中出现过，那么数据库管理系统就不会将这条记录插入到数据表中。一般在为列创建唯一约束后，数据库会为该列建立一个唯一索引，其索引名与约束名是相同的。
- ❑ 主索引：在数据表中使用PRIMARY KEY关键字可以为一个数据列定义一个主索引。所谓主索引，就是在定义的主键列中创建的索引。主索引也保证了数据列中记录的唯一性。一般在为列创建主键约束后，数据库会为该列建立一个主索引，其索引名与约束名是相同的。
- ❑ 单列索引：定义在数据表中一个数据列上的索引就是单列索引。一般在数据查询时，如果WHERE子句中经常用到数据表中的某一列作为查询条件，为了提高查询的效率，可以为该列创建单列索引。
- ❑ 复合索引：索引可以定义在一个数据表的多个数据列上，像这样的索引被称为复合索引。一般在数据查询时，如果WHERE子句中经常用到数据表中的某几个列作为查询条件，为了提高查询的效率，可以为这多个列创建复合索引。
- ❑ 聚簇索引：为了提高SQL语句对数据表的查询效率，可以为数据表创建一个聚簇索引。聚簇索引中索引项的顺序与数据表中数据记录的物理顺序保持一致。聚簇索引在每一个数据表中只能创建一个。

#### 注意

在数据表中，既可以在单列上创建索引，也可以在多列上创建索引。同一张数据表中也可以建立多个索引。一般数据库中对一张数据表中创建的索引数量给予限制。例如在MySQL数据库中，同一张数据表创建的索引总数不能超过16个；在Oracle数据库中，如果要创建复合索引，则索引列的总数不能超过32个。

### 4.4.2 创建与删除索引

在数据表中，创建和删除索引一般是由数据库管理员或者是数据表的创建者来完成的。如果想创建索引，可以使用CREATE INDEX语句完成。创建索引的语法格式如下：

```
CREATE [UNIQUE] | [CLUSTER] INDEX 索引名  
ON 表名 (列名 [排序方式]...)
```

其中，关键字UNIQUE表示创建的索引是唯一索引；关键字CLUSTER表示创建的索引是聚簇索引；这两个索引是可选的。关键字INDEX后面跟的是要创建的索引的名字；关键字ON用来指定索引要创建在哪一张数据表的哪一个或者哪几个列中；关键字ON后面跟的是数据表的名字；表名后面的括号里是用来指定索引要定义在哪一个或者哪几个列中；排序方式表示指定建立索引的排序是升序还是降序排列，关键字ASC表示升序排序，关键字DESC表示降序排序，默认情况下是升序排序。列名后面的排序方式是可选的。

例4.7 为教师信息表中表示教师职称单列创建索引。

```
CREATE INDEX i_profession  
ON T_teacher(profession)
```

这段SQL语句是为教师信息表中教师职称的列profession创建索引。其中，i\_profession表示索引的名字；T\_teacher(profession)表示要为教师信息表T\_teacher的列profession创建索引。

选中MySQL 5.0用户图形界面的右侧Schemata选项下的test\_STInfo数据库中的教师信息表T\_teacher，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，在出现的表编辑对话框中，选择对话框下方的“Indices”选项卡，可以看到为教师信息表T\_teacher中列profession创建的索引信息，如图4.7所示。

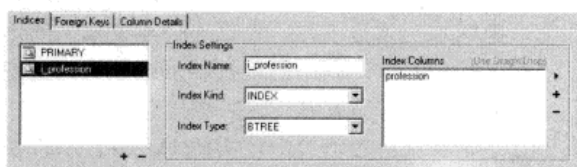


图4.7 单列索引信息

在图4.7中，“Indices”选项卡下描述了对教师信息表T\_teacher中列profession创建的单列索引信息。在“Index Settings”选项下指定了数据表T\_teacher中单列约束的信息。其中，“Index Name”指定了索引的名字，“Index Kind”指定了索引的类型，这里的索引类型设定为“INDEX”，“Index Type”指定了索引类型；在界面的右下方空白处，在Index Columns选项下指定了唯一索引的列名为profession。

在数据表中，除了可以为数据表中的某一列创建单列索引外，也可以为数据表的多个列创建复合索引。例如下面这个例子。

例4.8 为教师信息表中表示教师所在院系的列和教师职称的列创建复合索引。

```
CREATE INDEX i_dept_profession  
ON T_teacher(dept,profession)
```

这段SQL语句是为教师信息表中教师所在院系的列和教师职称的列创建复合索引。其中，i\_dept\_profession表示索引的名字；T\_teacher (dept, profession)表示要为教师信息表T\_teacher的列dept和列profession创建索引。列dept表示教师所在的院系，列profession表示教师职称。

选中MySQL 5.0用户图形界面的右侧Schemata选项下的test\_STInfo数据库中的教师信息表T\_teacher，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，在出现的表编辑对话框中，选择对话框下方的“Indices”选项卡，可以看到为教师信息表T\_teacher中列dept和列profession创建的复合索引的信息，如图4.8所示。

在图4.8中，“Indices”选项卡下描述了对教师信息表T\_teacher中列dept和列profession创建的复合索引信息。在“Index Settings”选项下指定了数据表T\_teacher中复合索引的信息。其中，“Index

Name”指定了索引的名字，“Index Kind”指定了索引的类型，这里的索引类型设定为“INDEX”，“Index Type”指定了索引类型；在界面的右下方空白处，在Index Columns选项下指定了复合索引的列名，分别为dept和profession。

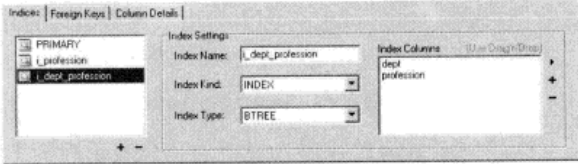


图4.8 复合索引信息

当然，在为数据表中的列创建索引时，也可以在列名的后面指定索引值的排序方式。例如，现在要为表示教师的工资字段创建一个索引，并且让索引只按照升序的方式排序，其SQL语句就可以使用如下的写法完成。

```
CREATE INDEX i_salary
ON T_teacher(salary ASC)
```

上面的两个例子中，分别在数据表中创建了单列索引和复合索引，当然也可以使用创建索引的语法格式中的UNIQUE和CLUSTER关键字为数据表创建唯一索引和聚簇索引。根据上面讲到的例子，读者很容易地就可以创建出唯一索引和聚簇索引。这里就不再举例了。

删除索引很简单，使用DROP INDEX语句就可以将一个索引删除。索引删除的同时，数据库管理系统会将数据字典中有关该索引的描述一并删除。删除索引的语法格式如下：

DROP INDEX 索引名

例如，现在想把教师信息表（T\_teacher）中为列教师职称创建的索引删除，就可以使用DROP INDEX来完成。

例4.9 删除教师信息表中为列教师职称创建的索引。

```
DROP INDEX i_profession
```

其中，i\_profession表示索引的名字。

**注意** 创建和删除索引一般是由数据库管理员或者是数据表的创建者来完成的。一般其他用户不能创建和删除索引，如果用户希望在数据表中创建和删除索引，必须要取得相应的管理权限。有关数据表中的权限授予的内容可以参看第15章。

## 4.5 修改数据库中的表

如果在创建完数据表之后，还需要对数据表结构做一些修改，例如向数据表中增加某一列、删除某一列、为数据表中的某一列增加约束条件或者索引、修改数据表中某一列的数据类型等，这个时候就需要使用ALTER TABLE语句。这一节就来介绍如何使用ALTER TABLE语句修改数据表结构。

### 4.5.1 向表中增加一列

在实际应用中，根据实际需要，有些时候需要为数据表增加一个指定的列，用来完善数据表信息。向数据表增加一列的语法格式如下：

```
ALTER TABLE table_name ADD(column_name datatype [constraint_condition])
```

其中，ALTER TABLE是修改表结构的关键字；关键字ALTER TABLE之后跟的table\_name是表的名字；ADD是关键字，用来指定数据表中要增加的列；关键字ADD后面的括号中需要指定新增加的列

零基础学SQL

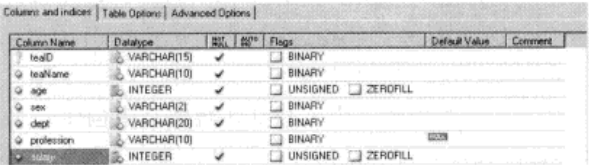
名、该列对应的数据类型和完整性约束条件。column\_name表示新增加的列名，datatype用来指定该列的数据类型，constraint\_condition用来表示该列的完整性约束条件，它是可选的。

例4.10 向教师信息表中增加一个表示教师工资的列。

```
ALTER TABLE T_teacher ADD salary INT NOT NULL
```

这段SQL是为教师信息表增加一个表示教师工资的列。其中，T\_teacher表示教师信息表的名字；salary表示教师信息表中新增加的列；INT指定该列的数据类型；NOT NULL指定列salary的完整性约束条件，这里为列salary定义一个非空约束。

选中MySQL 5.0用户图形界面的右侧Schemata选项下的test\_STInfo数据库中的教师信息表T\_teacher，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，在出现的表编辑对话框中，从“Columns and Indices”选项卡下的表格中可以看到，在该表格的最后多出了一个salary列，如图4.9所示。



Column Name	Datatype	PK	PK*	Flags	Default Value	Comment
teaID	VARCHAR(15)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	BINARY		
teaName	VARCHAR(10)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	BINARY		
age	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	UNSIGNED <input type="checkbox"/> ZEROFILL		
sex	VARCHAR(2)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	BINARY		
dept	VARCHAR(20)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	BINARY		
profession	VARCHAR(10)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	BINARY	0000	
salary	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	UNSIGNED <input type="checkbox"/> ZEROFILL		

图4.9 向数据表中增加一列

4.5.2 增加一个约束条件

如果在创建一个数据表时没有为指定的列定义约束条件，在数据表创建完成后，可以使用ALTER TABLE语句中的ADD子句为某一列增加一个约束条件。语法格式如下：

```
ALTER TABLE table_name ADD constraint_type (column_name)
```

其中，ALTER TABLE是修改表结构的关键字；关键字ALTER TABLE之后跟的table\_name是表的名字；ADD是关键字，用来指定数据表中要增加的列；关键字ADD后面的括号中需要指定约束条件和新增的列名。constraint\_type表示指定的约束条件，column\_name表示新增加的列名。

使用ALTER TABLE语句可以为数据表增加UNIQUE约束、PRIMARY KEY约束、CHECK约束。例如下面这个例子。

例4.11 为院系信息表增加主键约束。

```
ALTER TABLE T_dept ADD PRIMARY KEY(deptID)
```

这段SQL是为院系信息表中的列deptID增加主键约束。其中，T\_dept表示院系信息表的名字；PRIMARY KEY表示要为T\_dept表中的列增加主键约束，关键字PRIMARY KEY后的括号中指定了列的名字。列deptID表示院系编号。

读者可以选中MySQL 5.0用户图形界面的右侧Schemata选项下的test\_STInfo数据库中的院系信息表T\_dept，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，看一下院系信息表中表结构是否发生了变化。

在SQL语句中，除了可以为数据表增加UNIQUE约束、PRIMARY KEY约束和CHECK约束之外，也可以为数据表增加FOREIGN KEY约束。为数据表中的某一列增加FOREIGN KEY约束的方法与上面的方法有些不同。

增加FOREIGN KEY约束需要在ALTER TABLE语句的ADD子句之后增加一个REFERENCES关键字，用来指明与之关联的主表的表名和相应的列。增加FOREIGN KEY约束的语法格式如下：

```
ALTER TABLE table_name1 ADD FOREIGN KEY (column_name1)
REFERENCES table_name2(column_name2)
```

其中，table\_name1表示从表的名字；FOREIGN KEY是定义外键的关键字；column\_name1表示列名，用来指定数据表中用于外键约束条件的外键；REFERENCES关键字用来指定主表中的表名和主表中的关键列；table\_name2表示主表的名字；column\_name2表示与从表列column\_name1对应的主键列的名字。

例4.12 为成绩信息表增加外键约束。

```
ALTER TABLE T_result ADD FOREIGN KEY(curID)
REFERENCES T_curriculum(curID)
```

这段SQL语句是为成绩信息表中的列curID增加外键约束。其中，FOREIGN KEY表示创建外键约束的关键字；curID表示为成绩信息表（T\_result）中的表示课程编号的列curID定义外键约束；REFERENCES T\_curriculum(curID)表示将列curID定义为一个指向课程信息表T\_curriculum的主键curID中的外键。

选中MySQL 5.0用户图形界面的右侧Schemata选项下的test\_STInfo数据库中的成绩信息表T\_result，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，在出现的表编辑对话框中，选择对话框下方的“Foreign Keys”选项卡，可以看到为成绩信息表T\_result中列curID定义的外键约束信息，如图4.10所示。

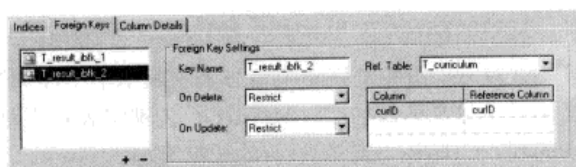


图4.10 成绩信息表外键约束信息

在图4.10中，“Foreign Key Settings”选项下描述了对数据表T\_result中列curID定义的外键约束信息。其中，“Key Name”表示主键的名字；“Ref.Table”指定了表T\_result对应的主表，这里表T\_result对应的主表为T\_curriculum（课程信息表）；On Delete指定了数据删除方式，这里的删除方式指定为Restrict（受限删除）；On Update指定了数据修改方式，这里的修改方式指定为Restrict（受限修改）。

### 4.5.3 增加一个索引

如果在创建一个数据表时没有为指定的列定义索引，在数据表创建完成后，可以使用ALTER TABLE语句中的ADD子句为列增加一个索引。其语法格式如下：

```
ALTER TABLE table_name ADD INDEX(column_name1[, column_name2]...)
```

其中，ALTER TABLE是修改表结构的关键字；关键字ALTER TABLE之后跟的table\_name是表的名字；ADD是关键字，用来指定数据表中要增加的列；INDEX关键字表示要为数据表增加一个索引。column\_name1表示为增加索引指定的列。指定列可以是一列，也可以是多列。

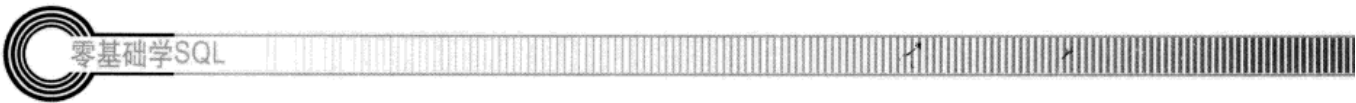
例4.13 为课程信息表中的列增加索引。

```
ALTER TABLE T_curriculum ADD INDEX i_credit(credit)
```

这段SQL语句是为课程信息表中的列credit增加索引。其中，T\_curriculum表示课程信息表的名字；i\_credit表示要增加的索引的名字；credit表示增加索引指定的列。这里列credit表示课程的学分。

选中MySQL 5.0用户图形界面的右侧Schemata选项下test\_STInfo数据库中的课程信息表T\_curriculum，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，在出现的表编辑对话框中，





选择对话框下方的“Indices”选项卡，可以看到为课程信息表T\_curriculum中列credit增加的索引信息，如图4.11所示。

**4.5.4 修改表中的某一列**

在实际应用中，有些时候需要对数据表中的某一列的数据类型、默认值等内容进行修改，这时就需要使用ALTER TABLE中的MODIFY子句。其语法格式如下：

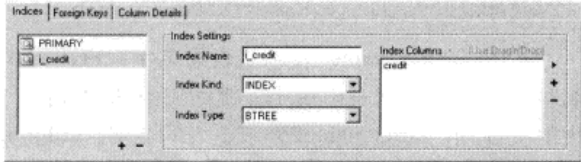


图4.11 为课程信息表增加索引信息

```
ALTER TABLE table_name MODIFY column_name datatype
```

其中，ALTER TABLE是修改表结构的关键字；关键字ALTER TABLE之后跟的table\_name是表的名字；MODIFY是关键字，用来表示要对数据表中的某一列进行修改操作；关键字MODIFY后面需要指定要修改列的名字和该列对应的数据类型；column\_name表示列的名字；datatype表示列的数据类型。

**例4.14 修改学生信息表中表示学生年龄列的数据类型。**

```
ALTER TABLE T_student MODIFY sex CHAR(2)
```

这段SQL语句是将学生信息表中表示学生性别的列的数据类型修改为CHAR类型。其中，T\_student表示学生信息表的名字；MODIFY 关键字表示要对指定列进行修改；sex是指学生信息表中表示学生性别的列；CHAR(2)表示列sex的数据类型。

选中MySQL 5.0用户图形界面的右侧Schemata选项下test\_STInfo数据库中的学生信息表T\_student，单击鼠标右键，在出现的列表中选择“Edit Tabel”选项，在出现的表编辑对话框中，从“Columns and Indices”选项卡下的表格中可以看到，列sex对应的数据类型已经由原来的VARCHAR类型变成了现在的CHAR类型，如图4.12所示。

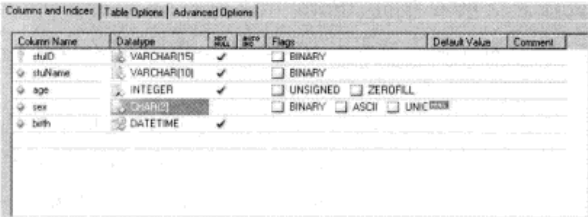


图4.12 修改学生信息表中表示学生年龄列的数据类型

**4.5.5 删除表中某一列**

如果在数据表中的某一列不需要使用了，可以使用ALTER TABLE语句中的DROP子句将该列删除。其删除的语法格式如下：

```
ALTER TABLE table_name DROP column_name
```

其中，ALTER TABLE是修改表结构的关键字；关键字ALTER TABLE之后跟的table\_name是表的名字；DROP是关键字，用来表示要对数据表中的某一列进行删除操作；关键字DROP后面需要指定要删除列的名字。

**例4.15 删除教师信息表中表示院系编号的列。**

```
ALTER TABLE T_teacher DROP deptID
```

这段SQL语句是将教师信息表中表示教师所在院系编号的列删除。其中，T\_teacher表示教师信息表的名字；DROP关键字表示要对指定列进行删除；deptID是指教师信息表中表示教师所在院系编号的列。



#### 4.5.6 删除一个约束条件

如果在数据表中将某一个列的约束条件删除，可以使用ALTER TABLE语句中的DROP子句。其删除约束条件的语法格式如下：

```
ALTER TABLE table_name DROP constraint_type
```

其中，ALTER TABLE是修改表结构的关键字；关键字ALTER TABLE之后跟的table\_name是表的名字；DROP是关键字，用来表示要将数据表中列的约束条件删除；关键字DROP后面需要指定要删除约束条件的名字；constraint\_type表示要删除的约束条件。

例4.16 删除院系信息表中的主键。

```
ALTER TABLE T_dept DROP PRIMARY KEY
```

这段SQL语句是将院系信息表中的主键约束删除。其中，T\_dept表示院系信息表的名字；DROP关键字表示要对列的约束条件进行删除；PRIMARY KEY表示主键约束。

#### 4.6 删除数据库中的表

如果在实际使用中某一个数据表已经不再需要时，可以使用DROP TABLE语句将其删除。其语法格式如下：

```
DROP TABLE table_name[CASCADE CONSTRAINTS]
```

其中，DROP TABLE是用于删除数据表的关键字；table\_name表示要删除的表的名字。使用DROP TABLE执行数据表的删除操作时，该表中的记录以及由此表建立的相应视图都会被同时删除掉。如果数据表之间存在外键约束，删除主表时要使用CASCADE CONSTRAINTS级联删除从表中的外键约束。它是可选的。一般用户如果要删除数据表，需要具有删除数据表的权限。（有关权限的授予的内容将在第15章介绍）

例4.17 删除教师信息表。

```
DROP TABLE T_teacher
```

这段SQL语句是要将教师信息表删除。教师信息表中的记录以及通过该表建立的相关视图也都会被同时删除。

如果想要删除的数据表和另外一张表中存在外键约束的条件，那么当要删除其中的主表时，需要在删除语句中使用CASCADE CONSTRAINTS。

例4.18 删除课程信息表。

```
DROP TABLE T_curriculum CASCADE CONSTRAINTS
```

#### 注意

如果只想删除数据表中的记录，需要使用DELETE语句。（有关DELETE语句的使用方法可以参看第14章）。

#### 4.7 数据库test\_STInfo中的表

在前面几节的讲解中，用到了学生信息表、成绩信息表、课程信息表、教师信息表、院系信息表



等。这些数据表都是在test\_STInfo数据库中创建的。这一节将对数据库test\_STInfo中涉及到的这些数据表的内容和表的基本结构做一个总结性的介绍。这些数据表在后面讲解SQL语句的查询和更新操作时会经常用到。

4.7.1 学生信息表T\_student

学生信息表（T\_student）中存储的是记录学生基本情况的信息，学生信息表中一共包括5个字段。其表结构如表4.3所示。

表4.3 学生信息表T\_student

字段名	数据类型	字段长度	关键字	是否为空
stuID	VARCHAR	15	是	否
stuName	VARCHAR	10	否	否
age	INT	2	否	否
sex	CHAR	2	否	否
birth	DATETIME	10	否	否

其中，stuID表示学生编号，stuName表示学生姓名，age表示学生年龄，sex表示学生姓名，birth表示学生的出生日期。列stuID作为该表的主键。

4.7.2 课程信息表T\_curriculum

课程信息表（T\_curriculum）中记录与学生选课有关的信息，课程信息表中一共包括5个字段。其表结构如表4.4所示。

表4.4 课程信息表T\_curriculum

字段名	数据类型	字段长度	关键字	是否为空
curID	VARCHAR	15	是	否
curName	VARCHAR	10	否	否
credit	INT	2	否	否
learnTime	INT	2	否	否
teacherName	VARCHAR	10	否	否

其中，curID表示课程编号，curName表示课程名称，credit表示课程学分，learnTime表示课程的学时，teacherName表示授课教师。列curID作为该表的主键。

4.7.3 成绩信息表T\_result

成绩信息表（T\_result）中记录的是学生选课的考试成绩，成绩信息表中一共包括3个字段。其表结构如表4.5所示。

其中，stuID表示学生编号，curID表示课程编号，result表示课程成绩。列stuID和列curID联合作为该表的主键。

表4.5 成绩信息表T\_result

字段名	数据类型	字段长度	关键字	是否为空
stuID	VARCHAR	15	是	否
curlID	VARCHAR	15	是	否
result	DOUBLE	5	否	是

4.7.4 教师信息表T\_teacher

教师信息表（T\_teacher）中存储的是记录教师基本情况的信息，教师信息表中一共包括9个字段。其表结构如表4.6所示。

表4.6 教师信息表T\_teacher

字段名	数据类型	字段长度	关键字	是否为空
teaID	VARCHAR	15	是	否
teaName	VARCHAR	10	否	否
age	INT	2	否	否
sex	CHAR	2	否	否
deptID	VARCHAR	15	否	否
dept	VARCHAR	20	否	否
profession	VARCHAR	10	否	是
salary	INT	5	否	否
pension	DOUBLE	5	否	是

其中，teaID表示教师编号，teaName表示教师姓名，age表示教师年龄，sex表示教师姓名，deptID表示教师所在院系编号，dept表示教师所在院系，profession表示教师职称，salary表示教师工资，pension表示教师津贴。列teaID作为该表的主键。

4.7.5 院系信息表T\_dept

院系信息表（T\_dept）是记录学校的院系信息，院系信息表中一共包括院系编号和院系名称两个字段的的信息。其表结构如表4.7所示。

表4.7 院系信息表T\_dept

字段名	数据类型	字段长度	关键字	是否为空
deptID	VARCHAR	15	是	否
deptName	VARCHAR	10	否	否

其中，deptID表示院系编号，deptName表示院系名称。列deptID作为该表主键。

4.7.6 计算机系教师信息表T\_CSteacher

计算机系教师信息表（T\_CSteacher）中存储的是记录计算机系教师基本情况的信息，计算机系教师信息表中一共包括9个字段。其表结构如表4.8所示。



表4.8 计算机系教师信息表T\_CSteacher

字段名	数据类型	字段长度	关键字	是否为空
teaID	VARCHAR	15	是	否
teaName	VARCHAR	10	否	否
age	INT	2	否	否
sex	CHAR	2	否	否
deptID	VARCHAR	15	否	否
dept	VARCHAR	20	否	否
profession	VARCHAR	10	否	是
salary	INT	5	否	否
pension	DOUBLE	5	否	是

其中，teaID表示教师编号，teaName表示教师姓名，age表示教师年龄，sex表示教师性别，deptID表示教师所在院系编号，dept表示教师所在院系，profession表示教师职称，salary表示教师工资，pension表示教师津贴。列teaID作为该表的主键。

### 4.8 小结

本章主要对数据表中涉及的主键、外键、约束和索引的概念以及在数据表中为列定义约束和索引的使用方法做了比较详细的介绍。另外还介绍了使用SQL语句创建和删除一个数据表的方法。

如果在创建完数据表之后，还需要对数据表结构做一些修改，需要使用ALTER TABLE语句。通过ALTER TABLE语句可以向数据表中增加或者删除一个指定的列，增加或者删除一个约束条件，增加一个索引，也可以修改表中的某一列，还可以删除指定的数据表。

在本章的最后一节对数据库test\_STInfo中涉及的数据表的内容和表的基本结构做了一个总结性的介绍。这些数据表在以后讲解SQL执行查询和更新操作时会经常用到。



## 第三篇 数据查询

### 第5章 基本查询操作

在数据库的操作中，开发人员或者用户为了取得数据记录，需要检索数据表中的信息。在SQL语句中，检索数据记录是通过SELECT语句来完成的。SELECT语句可以检索数据表或者视图中的数据，并将查询出来的数据以结果集的形式显示出来。从本章开始到第10章将对SELECT查询语句做全面的介绍，本章主要介绍SELECT语句查询的基本操作。

本章重点：

- ☐ 查询全部列的记录
- ☐ 查询表中指定的列
- ☐ 查询表中不重复的记录
- ☐ 使用列别名查询
- ☐ 对查询的记录进行算术运算
- ☐ 使用连接符（||）连接字段

#### 5.1 查询全部列的记录

在SQL语句中，如果想要检索数据表中全部列的记录，就需要对数据表中的所有列进行查询。在SQL语句中，提供了一种方便查询数据表或者视图的所有列的方法，其语法格式如下：

```
SELECT *  
FROM 表名或者视图名[,表名或者视图]
```

其中，SELECT语句后面的“\*”号表示查询数据表中的所有列，FROM子句后面的表名或者视图名用来表示指定要查询数据表或者视图的名字。[]里面指定的表名或者视图是可选的，也就是说，FROM子句后面可以跟多个表或者视图的名字，多个表或者视图之间用逗号分开。

**说明**

FROM子句后面最多可以指定256个表或者视图的名字，FROM子句后面如果要指定多个表名或者视图名，它们之间需要用逗号分开。

### 例5.1 查询学生信息表（T\_student）中所有学生的全部信息。

```
SELECT *  
FROM T_student
```

这里指定要查询学生信息表（T\_student）中所有学生的全部信息，所以需要查询学生信息表（T\_student）中所有的列。学生信息表（T\_student）中共有5列，分别是stuID、stuName、age、sex和birth。其查询结果如图5.1所示。

从显示的结果可以看到，其中，列stuID表示学生编号，列stuName表示学生的名字，列age表示学生的年龄，列sex表示学生的性别，列birth表示学生的出生日期。在MySQL 5.0数据库中，日期类型的变量在数据库中是以年-月-日 时：分：秒的形式显示的。

在实际的SELECT语句查询中，并不建议使用“\*”号查询表中所有的列，最好应该在SELECT语句中指定要查询列的名字。在SELECT语句中指定要查询列的名字主要基于以下两个方面的原因。

- ❑ 从软件开发的角度考虑，在实际的项目开发中，数据表中的字段一般都很多，在SELECT语句中指定要查询列的名字，有利于开发人员或者用户对数据表的了解，也有利于对数据表的维护。而使用“\*”号查询表中所有的列，开发人员或者用户就无法了解数据表中的结构，也不利于对数据表的维护。
- ❑ 从执行效率角度考虑，使用“\*”号查询表中所有的列的速度比指定查询列的名字的查询方式的速度慢。使用指定列的名字的查询方式可以提高查询的效率。

那么，既然应该在SELECT语句中使用指定查询列的名字的方式检索数据表中的信息，那么如何在SELECT语句中指定列的名字查询数据表中的记录呢？在5.2节中将向读者介绍查询数据表中指定列的方法。

## 5.2 查询表中指定的列

在实际开发应用中，很多时候，开发人员或者用户并不希望看到数据表中所有的记录，而只是对数据表中其中某一列或者某几列的数据感兴趣，此时就需要查询数据表中指定列的信息。在SELECT语句中查询数据表或者视图指定列的语法格式如下：

```
SELECT 目标列[,目标列,...]  
FROM 表名或者视图名[,表名或者视图]
```

其中，SELECT语句后面的目标列表示要查询的指定列的名字。[]里面指定的目标列是可选的。也就是说，SELECT语句中指定的目标列可以是一列也可以是多个列，指定多个列时，多个列名之间需要用逗号分开。FROM子句后面的表名或者视图名用来表示指定要查询数据表或者视图的名字。[]里面指定的表名或者视图是可选的，也就是说，FROM语句后面可以跟多个表或者视图的名字，多个表或者视图之间用逗号分开。

例5.2 查询学生信息表（T\_student）中学生编号以及学生编号对应的学生姓名信息。

```
SELECT stuID ,stuName  
FROM T_student
```

stuID	stuName	age	sex	birth
s102203	赵亮	23	男	1966-05-16 00:00:00
s112303	郑丽	21	女	1989-01-25 00:00:00
s115263	王博	23	男	1986-08-02 00:00:00
s206363	张明	22	男	1987-04-23 00:00:00
s221256	王昌辉	24	男	1985-03-18 00:00:00
s231456	王玉梅	22	女	1987-03-28 00:00:00
s232516	李玉峰	24	女	1985-08-16 00:00:00
s253263	李凤	24	女	1986-06-06 00:00:00

图5.1 查询学生信息表（T\_student）中所有学生的全部信息



这里指定要查询学生信息表（T\_student）中学生编号以及学生编号对应的学生姓名信息，所以只需要查询学生信息表（T\_student）中表示学生编号的stuID字段和表示学生姓名的stuName字段。在SELECT语句中就只指定了stuID和stuName。其查询结果如图5.2所示。

SELECT语句中指定列的次序是stuID、stuName。从显示的结果可以看到，图5.2中显示的查询结果是按照SELECT语句中指定列的次序给出的。

stuID	stuName
s102203	赵亮
s112303	郑茹
s115263	王海
s206363	张明
s221256	王昌辉
s231456	王宝梅
s232516	李玉峰
s253263	李凤

图5.2 查询学生信息表（T\_student）中学生编号以及学生编号对应的学生姓名信息

**注意** 在SELECT语句中查询数据表或者视图指定列时，在SELECT语句中指定的列名必须是指定的数据表或者视图中存在的列。

5.3 查询表中不重复的记录

在使用SELECT语句执行查询操作时，检索的是数据表中所有满足条件的行，如果数据表中有重复行也会被查询出来，例如下面这个例子。

例5.3 查询成绩信息表（T\_result）中选修课程的学生。

```
SELECT stuID
FROM T_result
```

这里要查询的是成绩信息表（T\_result）中选修课程的学生。在SELECT语句中通过成绩信息表（T\_result）中学生编号stuID字段查询选修课程的学生信息。其查询结果如图5.3所示。

图5.3中一共查询出了13条记录。而在这13条记录中很多学生编号的记录都是重复的。可以看到，这些重复的记录并不是用户所需要的，也没有什么实际的意义。

在实际应用中，往往不希望看到查询结果中有重复的记录行存在。为了在查询的结果中不显示重复的记录行，在SELECT语句中就需要加上一个DISTINCT关键字排除查询结果中重复行的记录。其语法格式如下：

stuID
s102203
s102203
s102203
s102203
s112303
s112303
s206363
s206363
s221256
s253263
s253263
s253263
s253263

图5.3 查询成绩信息表中（T\_result）选修课程的学生

```
SELECT DISTINCT目标列[,目标列,...]
FROM 表名或者视图名[,表名或者视图]
```

其中，SELECT语句后面的目标列表示要查询的指定列的名字。在目标列的前面加上一个DISTINCT关键字，表示指定目标列的查询结果中去掉重复的行。在SELECT语句中指定的目标列可以是一列也可以是多个列，指定多个列时，多个列名之间需要用逗号分开。FROM子句后面的表名或者视图名用来表示指定要查询数据表或者视图的名字。[]里面指定的表名或者视图是可选的，也就是说，FROM语句后面可以跟多个表或者视图的名字，多个表或者视图之间用逗号分开。

例5.4 查询成绩信息表（T\_result）中选修课程的学生（去掉重复行）。

```
SELECT DISTINCT stuID
FROM T_result
```

这里要查询的是成绩信息表（T\_result）中选修课程的学生，SELECT语句中表示学生编号stuID字段的前面加了一个DISTINCT关键字，表示查询的结果中不包括重复的学生记录。其查询结果如图5.4所示。

图5.4中一共查询出了5条记录。而在这5条记录中重复的学生编号的记录已经不存在了。可以看到，在显示的查询结果中已经不包括重复的学生编号的记录了。

stuID
s102203
s112303
s206363
s221256
s253263

图5.4 查询成绩信息表（T\_result）中选修课程的学生（去掉重复行）

5.4 使用列别名查询

在前面的SELECT语句显示的查询结果中，可以看到查询结果中显示的列的名字就是SELECT语句中指定的在数据表中定义的列的名字。这些在数据表中定义的列的名字一般都是英文。有时，为了更好地理解某一列显示的信息，在SELECT语句中可以使用列别名的形式改变查询结果中显示的列的名字。其语法格式如下：

```
SELECT 目标列 [AS] 列别名[, 目标列 [AS] 列别名...]
FROM 表名或者视图名[, 表名或者视图名]
```

其中，SELECT语句后面的目标列表示要查询的指定列的名字。AS关键字后面跟的就是要使用的列别名，其中关键字AS是可选的。在SELECT语句中指定的目标列可以是一列也可以是多个列，指定多个列时，多个列名之间需要用逗号分开。FROM子句后面的表名或者视图名用来表示指定要查询数据表或者视图的名字。[]里面指定的表名或者视图是可选的，也就是说，FROM语句后面可以跟多个表或者视图的名字，多个表或者视图之间用逗号分开。

例5.5 查询学生信息表（T\_student）中所有学生的全部信息（使用列别名）。

```
SELECT stuID AS 学生编号, stuName AS 学生姓名, age AS 年龄, sex AS 性别, birth AS 出生日期
FROM T_student
```

这里使用列别名的方式显示要查询学生信息表（T\_student）中所有学生的全部信息。其查询结果如图5.5所示。

从显示的查询结果可以看到，查询结果中的列标题都变成了中文的形式。这样显示的结果看起来更直观，也更容易理解。

学生编号	学生姓名	年龄	性别	出生日期
s102203	赵亮	23	男	1986-05-16 00:00:00
s112303	郑茹	21	女	1988-01-25 00:00:00
s115263	王海	23	男	1986-08-02 00:00:00
s206363	张明	22	男	1987-04-23 00:00:00
s221256	王昌鹤	24	男	1985-03-18 00:00:00
s231456	王玉梅	22	女	1987-03-28 00:00:00
s232516	李玉峰	24	女	1985-08-16 00:00:00
s253263	李兵	24	女	1985-06-05 00:00:00

图5.5 使用列别名的方式显示查询学生信息表（T\_student）中所有学生的全部信息

**说明** 在SELECT语句中使用列别名的形式对数据表或者视图查询时，可以在查询的列名后面使用一个空格代替关键字AS，空格后面再跟上列别名的名字。

在例5.5中，可以看到，为SELECT语句中查询的指定列使用列别名时并不需要使用单引号或者双引号将列别名引住，但是如果列别名中包含有空格或者特殊字符，例如点号（.）、逗号（,）、分号（;）、冒号（:）等，此时就需要使用单引号或者双引号将包含有空格或者特殊字符的列别名引起来。

例5.6 查询学生信息表（T\_student）中所有学生的全部信息（列别名中有空格）

```
SELECT stuID AS '学生 编号', stuName AS '学生 姓名'
FROM T_stude
```

这里在列别名中使用了空格，所以这里使用单引号将包含有空格的列别名引起来（也可以使用双引号）。其查询结果如图5.6所示。

**注意** 如果使用的列别名对字母大小写敏感，或者在使用包含有空格或者特殊字符的列别名，则必须使用单引号或者双引号将其引起来。如果不这样做，SQL语句在执行时就会报错。

学生编号	学生姓名
s102203	赵亮
s112303	郑茹
s115263	王海
s206363	张明
s221256	王昌鹤
s231456	王玉梅
s232516	李玉峰
s253263	李凤

5.5 对查询的记录进行算术运算

图5.6 列别名中包含空格的方式查询学生信息表中的学生信息

SELECT语句中还可以使用算术运算符对指定的列进行算术运算。其中算术运算符包括加（+）、减（-）、乘（×）、除（÷）。其中SELECT语句中乘除运算符的优先级高于加减运算符的优先级。通过使用算术运算可以取得所需要列的特定结果。

例5.7 查询教师的每年的工资的总收入。

```
SELECT teaID AS 教师编号,teaName AS 教师姓名, salary*12 AS 年收入
FROM T_teacher
```

这里要查询的是教师的年收入，在教师信息表（T\_teacher）中salary表示教师每个月工资的收入，所以要想取得教师每年的工资的总收入，需要将表示教师收入的salary字段值乘以12。同时这段SELECT语句中还为查询的teaID列、teaName列和salary列都指定了列别名。其查询结果如图5.7所示。

教师编号	教师姓名	年收入
t102225	赵伟	36000
t103265	张磊	45600
t105320	于波	33600
t106358	毛翠	48000
t156354	王新	30000
t156355	李中	50400
t181585	李慧	42000
t186585	孙立	38400

图5.7 查询教师的年收入

**说明** 在SELECT语句中对指定列使用算术运算符进行算术运算，只会改变显示的查询结果，并不会改变数据表中列的原有值。

5.6 使用连接符（||）连接字段

在使用SELECT语句查询的过程中，有时需要将两个或者是更多的字段连接起来显示一个更有意义的结果。在SELECT语句中，如果需要将多个字段连接起来，在Oracle数据库中可以使用“||”连接符来完成。下面来看一个使用“||”连接符的例子。

例5.8 将学生信息表（T\_student）中学生姓名和性别连接起来查询学生信息。

```
SELECT stuID , stuName|| sex
FROM T_student
```

这里使用“||”连接符将表示学生姓名的字段stuName和表示学生性别的字段sex连接起来，组成一个新的字符串。其查询结果如下所示。

stuID	stuName   sex
s102203	赵亮 男
s112303	郑茹 女

零基础学SQL

```
s115263    王海 男
s206363    张明 男
s221256    王昌鹤 男
s231456    王玉梅 女
s232516    李玉峰 女
s253263    李凤 女
```

**注意** 在使用“||”连接符连接多个字段时，其链接的数据类型应该是相同的。当连接不同的数据类型时，系统会报错。如果需要在连接的字段中加入字符或者是日期类型的值，需要将该字符值或者日期类型的值使用单引号引起来。

MySQL数据库和Microsoft SQL Server数据库不支持使用“||”连接符连接字段，所以在MySQL数据库和Microsoft SQL Server数据库中不能使用“||”连接符实行多个字段之间的连接。如果想连接多个字段，在MySQL数据库中使用CONCAT函数（有关CONCAT函数的使用方法可以参考10.1.17小节），在Microsoft SQL Server数据库中可以直接使用加号（+）运算符将多个字符串连接起来。

例如，在MySQL数据库中，如果想将学生信息表（T\_student）中学生姓名和性别连接起来查询学生信息，可以使用下面的SQL语句来完成。

```
SELECT stuID , CONCAT( stuName,sex )
FROM T_student
```

在Microsoft SQL Server数据库中，可以使用下面的SQL语句来完成。

```
SELECT stuID , stuName + sex
FROM T_student
```

## 5.7 关于NULL值

在数据库中，如果没有为该列赋值，而且该列没有默认值，此时查询的结果就为空值，即NULL。NULL既不表示空格，也不表示0。

**例5.9** 查询教师信息表（T\_teacher）中教师的津贴。

```
SELECT teaID ,teaName , profession,salary,pension
FROM T_teacher
```

这里要查询的是教师信息表（T\_teacher）中教师的津贴。其中，字段pension表示教师津贴。其查询结果如图5.8所示。

在T\_teacher表中，并不是所有的教师都有津贴。从查询的结果可以看到，表示教师津贴的字段pension所在的列中，如果教师没有津贴，则显示为NULL值。

在实际的开发应用中，在显示结果中显示NULL值并不是开发人员或者用户希望看到的。很多时候，并不希望将NULL值作为显示数据出现在查询的结果当中。例如，例5.9中，如果希望看到教师的总收入，那么，对于没有津贴的老师记录，教师的总收入将会显示为NULL值。

teaID	teaName	profession	salary	pension
t02225	赵伟	副教授	3000	260.5
t03265	张昌	教授	3800	300
t05320	于波	讲师	2900	223.8
t06398	毛翠	教授	4000	289.3
t156354	王新	讲师	2500	192.3
t156355	李中	教授	4200	310.2
t181585	李慧	教授	3500	276
t186585	孙立	讲师	3200	249.6

图5.8 查询教师信息表中教师的津贴

因此，在查询中就需要对查询结果中出现的NULL值进行处理。数据库中，都提供了对NULL值进行出来的函数。有关使用空值处理函数解决显示结果中出现NULL值的问题，可以参看10.6节。

## 5.8 小结

本章主要介绍的是SELECT查询语句中的基本查询操作。通过本章的学习，读者可以了解到使用SELECT语句如何查询数据表或者视图中的所有列以及指定列，每一个SELECT语句中都需要含有一个FROM子句。

在基本查询操作中还介绍了去除数据表中重复的列的方法，如何为列起别名，如何对查询的记录进行算术运算以及如何连接数据表中的字段等基本查询语句。但是仅仅使用简单的SELECT查询语句很多时候无法满足应用的需要，在以后的几章中将会逐步深入地讲解SELECT语句其他的查询功能。

## 第6章 使用WHERE子句查询表中满足条件的记录

在使用SQL语句进行查询操作时，很多时候开发人员或者用户并不是对数据表中的全部记录感兴趣，而只是想得到实际需要的数据记录，这时就需要对查询结果进行限制。在SQL语句中可以使用WHERE子句过滤掉不符合条件的记录。本章就介绍如何使用WHERE子句查询表中满足条件的记录。

本章重点：

- ☐ 算术比较运算符的使用
- ☐ BETWEEN...AND运算符查询指定条件范围的记录
- ☐ IN运算符查询与列表匹配的记录
- ☐ 字符串和时间的比较
- ☐ 逻辑查询中AND、OR和NOT运算符的使用
- ☐ 空值查询
- ☐ 字符匹配查询
- ☐ 转义字符的使用

### 6.1 比较查询

在WHERE子句中可以使用比较运算符对数值、字符值等信息进行查询。比较运算符这里归纳为三类：算术比较运算符、BETWEEN...AND运算符和IN运算符。这一节将分别对这三种比较运算符的查询方法进行介绍。最后还将介绍WHERE子句中字符串和时间的比较方法。

#### 6.1.1 算术比较运算符

SQL语句中的算术比较运算符主要包括=（等于）、>=（大于等于）、<=（小于等于）、>（大于）、<（小于）、!=（不等于）、<>（不等于）、!>（不大于）、!<（不小于）。在SELECT语句的WHERE子句中可以使用算术比较运算符对指定列进行比较，其语法格式如下：

字段1比较运算符 值

其中，字段1表示数据表中需要查询的字段列名，字段1后面跟的是算术比较运算符，值表示的是指定列要比较的数值。使用比较运算符返回的结果是一个逻辑值。如果逻辑值为TRUE，则会返回查询到的记录，如果逻辑值为FALSE，则不会返回相应的查询结果。

**说明** 在Microsoft SQL Server数据库中，使用“<>”符号表示不等于运算。

**例6.1** 查询课程表中课程学分>=4的课程信息。

```
SELECT curName,credit
```



```
FROM T_curriculum
WHERE credit>=4
```

其中，curName表示课程的名字，credit表示课程的学分。在WHERE子句中使用>=（大于等于）运算符将课程表中课程学分>=4的课程信息查询出来。其查询结果如图6.1所示。

curName	credit
计算机系统结构	4
数据库基础	5
C语言	6
高等数学	4

图6.1 查询课程表中课程学分>=4的课程信息

在这个查询结果中，符合查询条件的记录一共有4条。从这4条记录中可以看到，这4门课程对应的学分数都大于或者等于4。下面再看一个在字符值中使用比较运算符的例子。

例6.2 查询教师信息表中教师职称不是教授的教师信息。

```
SELECT teaID,teaName,dept,profession
FROM T_teacher
WHERE profession!='教授'
```

其中teaID表示教师编号，teaName表示教师的姓名，dept表示教师所在的部门，profession表示教师的职称。在WHERE子句中使用!=（不等于）运算符查询教师信息表中教师职称不是教授的教师信息。其查询结果如图6.2所示。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t105320	于波	计算机系	讲师
t156354	王新	数学系	讲师
t186585	孙立	物理系	讲师

图6.2 查询教师信息表中教师职称不是教授的教师信息

在这个查询结果中，符合查询条件的记录一共有4条。从上面的两个例子可以看出，在使用比较运算符执行SQL语句操作时，比较运算符的左侧和右侧的数据类型是可以相互兼容的。

例如在例6.1中WHERE子句的查询条件是课程学分>=4，这里的课程学分在数据表中定义的就是一个整数，在例6.2中WHERE子句的查询条件是profession!='教授'，这里的教师职称在教师信息表中定义的是字符类型。也就是说，在使用比较运算符时，不能让一个字符类型的数据进行类似大于或者小于一个整型数据的比较。例如下面的语句是完全错误的。

```
profession>=4
```

因为profession是一个字符类型，而4是一个整型，这两个数据类型是不能相互兼容的，也没有任何的可比性。当然，这种写法本身也违背了SQL的语法规则。

注意

在SQL语句中，如果在WHERE子句中比较的是数值型数据，则可以不使用单引号（例如，例6.1）；如果在WHERE子句中比较的是其他的数据类型（例如，字符串、日期型等）则必须使用单引号将其引住（例如，例6.2）。另外，WHERE子句中比较运算符的左侧和右侧的数据类型必须是类型兼容的。

提示

在SQL语句中提供了两个比较运算符来描述不等于运算。分别是!=（不等于）和<>（不等于），这两个运算符在使用时并没有太大的区别，可以互换使用。读者可以使用<>（不等于）运算符来重新执行一次例6.2，看一下查询的运行结果。

6.1.2 BETWEEN...AND运算符查询指定条件范围的记录

BETWEEN...AND运算符可以用来查询指定条件范围的记录。使用BETWEEN...AND运算符查询时

在BETWEEN运算符和AND运算符后面都需要给定一个值。其语法格式如下：

字段1 BETWEEN 值1 AND 值2

其中，字段1表示数据表中需要查询的字段；值1为给定数值中较小的值；值2为给定数值中较大的值。其最终查询的结果也包括值1和值2本身。来看下面这个例子。

例6.3 查询教师信息表中年龄在30~50岁之间的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
WHERE age BETWEEN 30 AND 50
```

其中teaID表示教师编号，teaName表示教师的姓名，age表示教师年龄，sex表示教师性别，dept表示教师所在的部门，profession表示教师的职称。WHERE子句中指定了要查询教师的年龄范围是在30~50岁之间。其查询结果如图6.3所示。

teaID	teaName	age	sex	dept	profession
1102225	赵伟	30	男	计算机系	副教授
1103265	张晶	43	男	计算机系	教授
1106358	毛草	50	女	计算机系	教授
1156354	王新	33	女	数学系	讲师
1181585	李慧	40	女	物理系	教授
1185585	孙立	48	男	物理系	讲师

图6.3 查询教师信息表中年龄在30~50岁之间的教师信息

这条SQL语句是从教师信息表中将教师年龄在30~50岁之间（包括30岁和50岁）的教师信息全都查询了出来。从查询的结果中可以看到，使用BETWEEN...AND运算符查询指定条件范围的记录时，其查询结果与age>=30 AND age<=50中得到结果相同。

#### 提示

在SQL语句中使用BETWEEN...AND运算符可以查询指定条件范围的记录，也可以使用NOT BETWEEN...AND运算符来排除一些记录。例如如果要查询教师信息表中年龄不在30~50岁范围之间的教师信息，就可以使用NOT BETWEEN...AND运算符来完成。读者可以试着自己来完成这个SQL语句。

### 6.1.3 IN运算符查询与列表匹配的记录

IN运算符用来查询与列表匹配的记录。使用IN运算符，可以将满足列表中满足指定表达式的任何一个值都查询出来。IN运算符后的属性值可以是一个，也可以有多个，多个属性值之间需要用逗号分隔。其语法格式如下：

字段1 IN(属性值1, 属性值2, 属性值3...)

其中字段1表示数据表中需要查询的字段；属性值1, 属性值2, 属性值3分别表示需要查询的值。属性值既可以是数字类型的也可以是字符类型的。如果属性值是字符类型的值，则需要使用单引号将其引住。

例6.4 查询教师信息表中教师所在部门为计算机系或者数学系的所有教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
WHERE dept IN('计算机系','数学系')
```

WHERE子句中的IN运算符后面的括号中指定要查询的是计算机系或者数学系的所有教师信息。其查询结果如图6.4所示。

这条SQL语句是从数据表中选择教师所在部门为计算机系或者数学系的教师信息，如果在教师信息

表中有计算机系或者数学系，就将其查询结果所在的行返回。在这个查询结果中，符合查询条件的记录一共有6条。这6条记录包括了部门为计算机系和数学系的所有教师信息。

提示

在SQL语句中可以使用IN运算符查询与列表匹配的记录，也可以使用NOT IN运算符来排除一些记录。例如如果要查询教师信息表中不在计算机系或者数学系的所有教师信息，就可以使用NOT IN运算符来完成。读者可以试着自己来完成这个SQL语句。

teaID	teaName	age	sex	dept	profession
t102225	赵伟	38	男	计算机系	副教授
t103265	张磊	43	男	计算机系	教授
t105320	于波	28	男	计算机系	讲师
t106358	毛翠	50	女	计算机系	教授
t156354	王新	33	女	数学系	讲师
t156355	李中	55	女	数学系	教授

图6.4 查询教师信息表中教师所在部门为计算机系和数学系的所有教师信息

使用IN运算符的查询语句可以用等号运算符和OR运算符（参看6.2.2小节）来替代，其查询结果是相同的。例如例6.4可以使用下面的SQL语句替换。

```
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
WHERE dept='计算机系' OR dept='数学系'
```

这个SQL语句中，将IN运算符的查询语句用OR运算符取代，但是如果需要查询的记录满足多个任一条件时，使用OR运算符SQL语句就不是很容易阅读。试想一下，如果要查询教师信息表中所在部门在计算机系、数学系、物理系、机械系的教师信息，使用OR运算符SQL语句书写起来可能就会比较烦琐，不如使用IN运算符的SQL语句简练和容易阅读。

6.1.4 字符串比较

在使用SQL语句进行比较查询时，经常会遇到字符串比较问题。对字符串进行比较时，常用的数据库都可以使用比较运算符对字符串进行比较，另外，在MySQL数据库中还可以使用关键字BINARY对字符串进行二进制比较。在这一小节中将对这两种字符串比较方式进行介绍。

1. 使用比较运算符对字符串进行比较

在使用比较运算符比较字符值时，比较运算符中左右两侧的字符值应该用单引号将该字符引住。下面来看一个使用比较运算符对字符串进行比较的例子。

例6.5 使用比较运算符对字符串进行比较。

```
SELECT 'sql5.0'>'sql'
```

这条SELECT语句是对字符串sql5.0和字符串sql进行比较，这里使用大于运算符（>）对两个字符串进行比较，其比较的结果如下所示。

```
+-----+
| 'sql5.0'>'sql' |
+-----+
|          1          |
```

其查询的结果为1，表示返回的结果为TRUE。即当在SELECT语句对指定的两个字符串进行比较时，认为字符串sql5.0确实比字符串sql大。

**说明** 在使用比较运算符对字符串进行查询和比较时，有的数据库需要区分字符的大小写，例如，Oracle数据库等。有的数据库对字符串进行查询和比较时不区分字符的大小写，例如，Microsoft SQL Server数据库和MySQL数据库等。为了避免由于不注意字符的大小写区分而查询不到正确的结果，可以使用将字符串全部转换为大写的UPPER函数和字符串全部转换为小写的LOWER函数对字符串进行大小写转换，有关这一部分内容将在10.1.2和10.1.3节中介绍。

在MySQL数据库中，对字符串的比较一般不区分大小写。例如下面这条SELECT语句。

```
SELECT 'mysql'='MySQL'
```

这条SELECT语句是对字符串mysql和字符串MySQL进行比较，由于MySQL数据库中对字符串的比较对字母的大小写不敏感，因此在使用等号运算符对这两个字符串进行比较时，数据库会认为这两个字符串是相等的，其查询的结果如下所示。

```
+-----+
| 'mysql'='MySQL' |
+-----+
|          1      |
+-----+
```

可以看到，查询的结果为1，表示返回的结果为TRUE。即当在SELECT语句对指定的两个字符串进行比较时，并不对字符串中字母的大小写进行区分。

## 2. 使用BINARY关键字对字符串进行二进制比较

如果希望比较的字符串区分大小写，可以使用关键字BINARY对字符串进行二进制比较。使用BINARY关键字，会把一个字符串（或者一个数字）转换为一个二进制的对象。

使用BINARY关键字对字符串进行二进制比较有两种语法格式，这两种语法格式是等价的。其语法格式如下：

```
SELECT string1 比较运算符 BINARY string2
SELECT BINARY string1 比较运算符 string2
```

其中，string1和string2表示要比较的两个字符串；比较运算符可以是=（等于）、>=（大于等于）、<=（小于等于）、>（大于）、<（小于）、!=（不等于）、<>（不等于）、!>（不大于）、!<（不小于）；BINARY是关键字，用来告诉数据库管理系统，需要对字符串进行二进制的比较。

下面通过一个例子来看一下使用BINARY关键字对字符串进行二进制比较时，查询的结果会有什么不同。

### 例6.6 使用BINARY关键字对字符串进行二进制比较。

```
SELECT 'mysql' = BINARY 'MySQL'
```

这条SELECT语句是对字符串mysql和字符串MySQL进行二进制比较，因为这段SQL语句中多了一个BINARY关键字，其查询的结果如下所示。

```
+-----+
| 'mysql' = BINARY 'MySQL' |
+-----+
|          0              |
+-----+
```

可以看到，查询的结果为0，表示返回的结果为FALSE。即当在SELECT语句对指定的两个字符串

进行二进制比较时，对字符串中字母的大小写进行了区分。

### 6.1.5 日期时间的比较

在WHERE子句中对日期值和时间进行比较时，要比较的日期和时间必须是数据库服务器可以接受的字符串格式。例如，在学生信息表（T\_student）中，学生的出生日期被设置为DATETIME日期类型的变量。要想在WHERE子句中对学生的出生日期值进行比较，可以使用单引号将该日期值引住。下面来看一个日期值比较的例子。

**例6.7** 查询学生信息表中出生日期在1986年01月01日之后的学生信息。

```
SELECT stuID ,stuName,age,sex,birth
FROM T_student
WHERE birth >'19860101'
```

这段SQL语句是查询学生信息表中出生日期在1986年01月01日之后的学生信息。在WHERE子句中给出了一个限定日期值的查询条件，其查询结果如图6.5所示。

从显示的结果可以看到，MySQL 5.0数据库中，日期类型的变量在数据库中是以年-月-日 时：分：秒的形式显示的。

stuID	stuName	age	sex	birth
s102203	赵亮	23	男	1986-05-16 00:00:00
s112303	郑磊	21	女	1988-01-25 00:00:00
s115263	王海	23	男	1986-08-02 00:00:00
s206363	张明	22	男	1987-04-23 00:00:00
s231456	王玉梅	22	女	1987-03-28 00:00:00

图6.5 查询学生信息表中出生日期在1986年01月01日之后的学生信息

**注意** 比较日期值和时间时需要使用单引号将其引住。

除了可以使用算术比较运算符对日期进行比较以外，也可以使用BETWEEN...AND运算符来查询指定某一个时间范围的记录。例如要查询出生日期的起始时间在1986年01月01日，终止时间在1988年12月12日的全体学生的记录，使用BETWEEN...AND运算符就可以这样写。

```
SELECT stuID ,stuName,age,sex,birth
FROM T_student
WHERE birth BETWEEN '19860101' AND '19881212'
```

在Oracle数据库中，日期比较一种更好的方法是使用日期转换函数TO\_DATE；在MySQL数据库中，则可以使用DATE\_FORMAT函数。通过这些函数可以将日期类型的值转换为需要的形式。TO\_DATE函数以及DATE\_FORMAT函数的使用方法将在10.3.2节中介绍。

## 6.2 逻辑查询

在SQL语句中逻辑运算符主要包括AND、OR和NOT三种。其中AND运算符用来查询同时满足多个条件的记录，OR运算符用来查询多个条件中满足其中一个条件的记录，NOT运算符用来查询满足相反条件的记录。这一节将对这三个逻辑运算符的使用方法一一介绍，并在最后通过例子介绍混合使用上述三种逻辑运算符进行复杂逻辑查询的情况。

### 6.2.1 使用AND运算符查询同时满足多个条件的记录

在SQL的执行操作中，很多情况下，WHERE子句并不是只希望满足一个条件，而是希望最终查询



的结果必须同时满足多个条件（两个或者两个以上）。这个时候就需要使用AND运算符。其语法格式如下：

条件1 AND 条件2

其中条件1、条件2是在WHERE子句中进行查询时都需要满足的条件。如果希望使用AND运算符在WHERE子句中连接多个条件，可以使用下面的语法格式。

条件1 AND 条件2 AND 条件3 ...

这里使用两个AND运算符来连接3个条件。多个AND运算符进行连接操作时，每一个AND运算符两侧的值必须都为TRUE，也就是说这些条件都同时被满足的情况下，结果才会被显示出来。

**说明** 在Microsoft SQL Server数据库中，使用“&”符号代替AND运算符表示逻辑与运算。

下面来看一个使用AND运算符的例子。在这个例子中使用AND运算符查询教师信息表中年龄在45岁以上并且职称为教授的教师信息。

**例6.8** 查询教师信息表中年龄在45岁以上并且职称为教授的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
WHERE age>45 AND profession='教授'
```

在这个查询语句中，查询的教师信息有两个条件的限制，一个条件是年龄在45岁以上，另外一个条件是职称为教授，而且这两个条件必须同时具备，缺一不可。因此这里就需要使用AND运算符将这两个查询条件连接起来。其查询结果如图6.6所示。

teaID	teaName	age	sex	dept	profession
1106358	毛翠	50	女	计算机系	教授
1156395	李中	55	女	数学系	教授

图6.6 查询教师信息表中年龄在45岁以上，职称为教授的教师信息

这条SQL语句是从教师信息表中选择教师年龄大于45岁并且教师职称为教授的教师信息。这里的SQL语句中AND运算符两侧的条件必须都为TRUE。可以看到age>45和profession='教授'这两个条件都符合SQL的语法规范而且比较的结果也都是TRUE。

**注意** SQL语句中AND运算符两侧的条件必须都为TRUE。如果在AND运算符两侧有任意一个条件为FALSE，则都不会显示正确的结果。

在这个查询结果中，符合查询条件的记录一共有2条。从这两条查询语句的结果可以看出，查询的教师年龄既大于45岁而且职称还都是教授。

**提示** AND运算符可以在WHERE子句中连接两个或者多个条件。例如现在要查询教师信息表中年龄在45岁以上，所在部门为计算机系的职称为教授的教师信息，这个时候最后查询出来的记录应该就只有一条了，这条查询语句读者可以试着完成一下。看一下最后的运行结果是不是只显示了一条记录。

## 6.2.2 使用OR运算符查询满足任一条件的记录

在使用SQL进行查询操作时，有些时候只是希望查询的结果中满足多个条件中的任一条件即可。这个时候就需要使用OR运算符。使用OR运算符可以用来查询满足任一条件的记录。其语法格式如下：



#### 条件1 OR条件2

其中条件1、条件2是在WHERE子句中进行查询时需要的条件。这两个条件中，只要符合其中任何一个条件，则符合该条件的记录就会被检索出来。如果希望使用OR运算符在WHERE子句中连接多个条件，可以使用下面的语法格式。

条件1 OR条件2 OR条件3 ...

这里使用两个OR运算符来连接3个条件。多个OR运算符进行连接操作时，两侧的条件中任何一个条件为TRUE，满足该条件的记录就会被显示出来。

**说明** 在Microsoft SQL Server数据库中，使用“|”符号代替OR运算符表示逻辑或运算。

下面来看一个使用OR运算符的例子。在这个例子中使用OR运算符查询教师信息表中年龄在45岁以上或者职称为教授的教师信息。

**例6.9** 查询教师信息表中年龄在45岁以上或者职称为教授的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
WHERE age>'45' OR profession='教授'
```

在这个查询语句中，查询的教师信息需要有两个，一个条件是年龄在45岁以上，另外一个条件是职称为教授，但是这两个条件不必同时具备，具备其中任何一个条件的记录都会被查询出来。其查询结果如图6.7所示。

这条SQL语句是从教师信息表中选择教师年龄大于45岁或者教师职称为教授的教师，并将这两部分的教师信息都查询出来。在这个查询结果中，符合查询条件的记录一共有6条。

从查询的结果中可以看到，教师编号为t103256的姓名为张昌的教师年龄是43岁不符合WHERE子句中age>'45'的条件，但是该教师的职称为教授，即符合WHERE子句中profession='教授'这个条件；而教师编号为t186585的姓名为张立的教师职称是讲师不符合WHERE子句中profession='教授'这个条件，但是他的年龄为48岁，即满足WHERE子句中age>'45'的条件。

从这里可以看出，在执行SQL语句操作中，如果OR运算符两侧的条件中有一个条件为TRUE，则满足该条件的记录就会被显示出来。

teaID	teaName	age	sex	dept	profession
t103256	张昌	43	男	计算机系	教授
t106358	毛翠	50	女	计算机系	教授
t156355	李中	55	女	数学系	教授
t181585	李慧	40	女	物理系	教授
t186585	孙立	48	男	物理系	讲师

图6.7 查询教师信息表中年龄在45岁以上或者职称为教授的教师信息

OR运算符可以在WHERE子句中连接两个或者多个条件。使用OR运算符时，OR运算符两侧有一个条件为TRUE时，满足该条件的记录就会被显示出来。所以OR运算符在使用时两侧条件可以只有一个为TRUE，也可以两侧的条件都为TRUE。

### 6.2.3 使用NOT运算符查询满足相反条件的记录

有些时候，需要查询不满足指定条件的记录，这个时候就需要使用NOT运算符。NOT运算符是用来查询满足相反条件的记录。

**说明** 在Microsoft SQL Server数据库中，使用“~”符号代替NOT运算符表示逻辑非运算。

下面来看一个使用NOT运算符的例子。在这个例子中使用NOT运算符查询教师信息表中不在计算机系或者数学系的所有教师信息。

**例6.10** 查询教师信息表中不在计算机系或者数学系的所有教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
WHERE dept NOT IN('计算机系','数学系')
```

其查询结果如图6.8所示。

这条SQL语句是将所有不在计算机系和数学系的教师信息检索出来。可以将该运行结果和例6.4中显示的结果做一下对比，看一下这两个SQL语句产生结果的不同之处。

teaID	teaName	age	sex	dept	profession
t181585	李慧	40	女	物理系	教授
t186585	孙立	48	男	物理系	讲师

图6.8 查询教师信息表中不在计算机系或者数学系的所有教师信息

### 6.2.4 复杂逻辑查询

在前面的3节中，分别介绍了AND运算符、OR运算符和NOT运算符的使用。这3个逻辑运算符也可以放到一个SQL语句中混合使用。在这3个逻辑运算符中，NOT的优先级最高，AND的优先级要高于OR。为了便于理解，一般在混合使用这3个逻辑运算符时，可以使用括号将每一个部分括起来。下面来看一个混合使用逻辑运算符的例子。

**例6.11** 查询计算机系或者是数学系中教师的工资大于3000元的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession ,salary
FROM T_teacher
WHERE (dept ='计算机系'OR dept ='数学系')
AND salary >3000
```

在这个查询语句中，使用了两个逻辑运算符，一个是OR，一个是AND。使用OR运算符查询所在的部门是计算机系或者是数学系的教师，使用AND运算符表示教师所在的部门是计算机系或者是数学系并且工资大于3000元，其查询结果如图6.9所示。

teaID	teaName	age	sex	dept	profession	salary
t103265	张昌	43	男	计算机系	教授	3800
t106358	毛翠	50	女	计算机系	教授	4000
t156355	李中	55	女	数学系	教授	4200

图6.9 查询计算机系或者是数学系中教师的工资大于3000的教师信息

这条SQL语句是从教师信息表中首先查询所在部门为计算机系或者是数学系的教师信息，将计算机系或者是数学系的教师信息的记录检索出来后，再在这些教师中查询工资大于3000元的教师信息，并将查询到的记录显示出来。

## 6.3 空值查询

在使用SQL语句执行查询操作时，还有一种是空值查询。在数据表中，如果一个表的行属性中不存在任何值的时候，也就是说该表的行属性中没有任何数据记录。那么就将其称之为空值。在SQL的查询中，NULL可以用来表示空值的含义。在SQL语句中，可以使用IS NULL或者IS NOT NULL关键字来判断空值。

**例6.12** 查询教师信息表中教师津贴为空的教师信息。

```
SELECT teaID ,teaName , profession,salary,pension
FROM T_teacher
WHERE pension IS NULL
```

第 6 章 使用WHERE子句查询表中满足条件的记录

这段SQL语句是查询教师信息表中教师津贴为空的教师信息。在WHERE子句中使用IS NULL关键字作为限定条件，用来表示教师津贴的字段pension所在列的值为空。其查询的结果如图6.10所示。

从上面的查询结果可以看到，查询出来的教师记录中其表示津贴的列pension的值都为NULL。如果想要查询教师津贴不为空的教师记录，可以使用IS NOT NULL关键字。例如对于例6.12，如果想要查询教师信息表中教师津贴不为空的教师信息，可以使用下面的SQL语句完成。

```
SELECT teaID,teaName,profession,salary,pension
FROM T_teacher
WHERE pension IS NOT NULL
```

这段SQL语句是查询教师信息表中教师津贴不为空的教师信息。在WHERE子句中使用IS NOT NULL关键字作为限定条件，用来限定表示教师津贴的字段pension所在列的值不为空。其查询的结果如图6.11所示。

teaID	teaName	profession	salary	pension
t105320	于波	讲师	2800	NULL
t156354	王新	讲师	2500	NULL
t186585	孙立	讲师	3200	NULL

图6.10 查询教师信息表中教师津贴为空的教师信息

teaID	teaName	profession	salary	pension
t102225	赵伟	副教授	3000	260.5
t103285	张磊	教授	3800	300
t106358	毛翠	教授	4000	289.3
t156355	李中	教授	4200	310.2
t181585	李慧	教授	3500	278

图6.11 查询教师信息表中教师津贴不为空的教师信息

从上面的查询结果可以看到，在教师信息表中凡是教师津贴不为空的教师信息全部都被检索了出来。

**注意** 当使用NULL查询数据表中的记录是否为空值时，不能使用比较运算符，只能使用IS NULL或者IS NOT NULL。使用比较运算符对NULL值进行比较时，其查询条件返回的结果始终是FALSE。也就是说，使用比较运算符对NULL值进行比较，不会检索出任何结果。

6.4 使用LIKE操作符实现模糊查询

在使用SQL语句进行查询时，经常会遇到这样一种情况，就是不能完全确定所需要查询信息的完整条件，但是这些信息又具有某些明显的特征。例如，想学习SQL语言，希望到图书馆中找一些相关的资料，但是又不知道有关SQL语言的书籍都有哪些，这个时候一般都会在图书管理系统中输入关键字SQL，这样与SQL有关的所有书籍就都会查到了。这就是模糊查询。在SQL语言中就提供了用于模糊查询的关键字LIKE，它需要和通配符“%”和“\_”配合使用。这一节就来介绍如何使用LIKE来实现模糊查询。

6.4.1 匹配任意单个字符

在SQL语句中，通配符“\_”表示匹配单个字符。即在查询语句中，一个“\_”只能表示匹配一个字符。下面来看一个使用“\_”的例子。

例6.13 查询学生信息表中7位学生编号中以s开头并以3结尾的学生信息。

```
SELECT stuID,stuName,age,sex,birth
FROM T_student
WHERE stuID LIKE 's____3'
```

例题明确需要查询的学生编号是7位，并且给定了其中第一位和最后一位的查询条件，因此需要使



用的通配符是“\_”。在WHERE子句中，使用了5个“\_”，每一个“\_”只能表示匹配学生编号中的任意一个字符。其查询结果如图6.12所示。

这条SQL语句是从学生信息表中选择学生编号为7位的并且以字符s开头以数字3作为学生编号的最后一位的所有学生信息。从查询的结果可以看到，所检索出来的学生编号都是7位，并且学生编号的开头字符都是s，结尾数字都是3。

除了使用LIKE和通配符“\_”进行匹配任意单个字符的模糊查询，也可以使用NOT LIKE和通配符“\_”查询不匹配任意单个字符的记录。来看下面这个例子。

**例6.14** 查询学生信息表中不匹配7位学生编号中以s开头并以3结尾的学生信息。

```
SELECT stuID,stuName,age,sex,birth
FROM T_student
WHERE stuID NOT LIKE 's_____3'
```

由于这里要查询学生信息表中不匹配7位学生编号中以s开头并以3结尾的学生信息，所以需要使用NOT LIKE关键字进行模糊查询，其查询的结果如图6.13所示。

从查询的运行结果可以看到，使用NOT LIKE关键字查询出来的结果和使用LIKE关键字查询出来的结果正好是相反的。

stuID	stuName	age	sex	birth
s102203	赵亮	23	男	1986-05-16 ...
s112303	郑茹	21	女	1988-01-25 ...
s115263	王海	23	男	1986-09-02 ...
s206363	张明	22	男	1987-04-23 ...
s253263	李凤	24	女	1985-06-06 ...

图6.12 查询学生信息表中7位学生编号中以s开头并以3结尾的学生信息

stuID	stuName	age	sex	birth
s221256	王昌鹤	24	男	1985-03-18 ...
s231456	王玉梅	22	女	1987-03-28 ...
s232516	李玉峰	24	女	1985-09-16 ...

图6.13 查询学生信息表中不匹配7位学生编号中以s开头并以3结尾的学生信息

**注意** 和字符串的比较一样，MySQL数据库中的LIKE操作符在进行匹配时，对字母的大小写不敏感。即在例6.13和例6.14中，LIKE操作符后面的字符s无论是大写还是小写，其查询的结果是一样的。

6.4.2 匹配0个或者多个字符

在SQL语句中，通配符“%”表示匹配0个或者多个字符。即一个“%”可以表示0个字符，也可以表示一个字符，还可以表示两个或者更多的字符。下面来看一个使用“%”的例子。

**例6.15** 查询学生信息表中姓王的所有学生的信息。

```
SELECT stuID,stuName,age,sex,birth
FROM T_student
WHERE stuName LIKE '王%'
```

这里使用的通配符是“%”。例题中需要查询的是学生信息表中姓王的所有学生的信息，而姓王学生的名字有的是两个字的，有的可能是三个字的，所以这里使用“%”通配符来匹配学生信息表中所有姓王的学生信息。其查询结果如图6.14所示。

这条SQL语句是从学生信息表中选择所有以“王”字开头的学生信息。这里的WHERE子句中的LIKE'王%'表示会匹配任何一个以“王”字开头的学生的信息。无论这个姓王的学生名字是两个字还是三个字，都会被检索出来。

stuID	stuName	age	sex	birth
s115263	王海	23	男	1986-09-02 ...
s221256	王昌鹤	24	男	1985-03-18 ...
s231456	王玉梅	22	女	1987-03-28 ...

图6.14 查询学生信息表中姓王的所有学生的信息

**注意** LIKE关键字后面的匹配字符必须要使用单引号。另外，在对字母进行匹配时，要注意区分字母的大小写。

## 第6章 使用WHERE子句查询表中满足条件的记录

同6.4.1节中“\_”通配符的使用一样，也可以使用NOT LIKE和通配符“%”查询不匹配0个或者多个字符的记录。读者可以参看例6.12自己试一下，这里就不再举例了。

**说明** 在Microsoft SQL Server数据库中除了支持“%”通配符和“\_”通配符之外，还支持使用“[]”和“[^]”通配符。其中通配符“[]”表示查询某一个范围内的所有单个字符，通配符“[^]”用来表示那些不在某一个指定范围内的字符。例如在Microsoft SQL Server数据库中，要想知道查询的字段是否匹配abcd，SQL语句可以这样写LIKE [abcd]（或者LIKE [abcd]）。同理，如果想匹配abcd以外的字符就可以使用LIKE [^abcd]（或者LIKE [^abcd]）。

### 6.4.3 使用转义字符

在使用SQL执行模糊查询时，有时数据表某个字段中的字符值本身就含有“%”或者是“\_”这两个字符。开发人员希望在查询时，将字符值本身就含有“%”或者是“\_”两个字符作为该字符的一部分查询出来，这个时候就需要使用ESCAPE关键字对其进行转义操作。使用ESCAPE关键字进行转义操作步骤如下：

- (1) 在需要转义的“%”或者是“\_”字符前加一个转义符，该转义符可以是一个任意字符。
- (2) 在ESCAPE关键字后指定该转义符的名称。

经过这两个步骤之后，位于该转义符之后的那个通配符（“%”或者是“\_”字符）就会被转义为一个普通字符。下面看一个使用转义字符的例子。

**例6.16** 查询院系信息表中院系编号对应的院系信息。

```
SELECT deptID,deptName
FROM T_dept
WHERE deptID like '%$_%' ESCAPE '$'
```

其中deptID表示院系编号，deptName表示院系名称。由于在院系信息表中院系编号是以字符\_数字的形式组合而成的，所以要想完整地输出院校编号，需要使用ESCAPE关键字对院系编号字段中的记录进行转义操作。其查询结果如图6.15所示。

deptID	deptName
L_10	计算机系
L_15	数学系
L_18	物理系

图6.15 查询院校信息表中院系编号对应的院系信息

从查询结果可以看到，包含有\_的院系编号字段信息完整地显示了出来。这里的转义符使用的是“\$”，也可以使用其他的字符。只要将该转义符放到需要转移到通配符的前面，并且使用ESCAPE关键字定义了转义符，就可以将该通配符转换为一个普通字段来处理。

**说明** 需要转义的“%”或者是“\_”字符前加的这个转义符可以是一个任意字符。例如可以是一个字母，也可以是一个“/”等。读者可以试一下，将例6.16中的“\$”符号换成字母a，看一下查询结果是否一样。

## 6.5 使用REGEXP关键字进行模式匹配

在MySQL数据库中，还提供了一种更加灵活的模式匹配方法，就是使用REGEXP关键字对字符串进行模式匹配。使用REGEXP关键字对字符串进行模式匹配，只要是在被匹配的字符串中含有与匹配



零基础学SQL

模板中相匹配的子串，就被认为是模式匹配。其返回值就为TRUE。

在MySQL数据库中，使用REGEXP关键字对字符串进行模式匹配时，可以使用一些模式匹配修饰符对模式匹配进行测试。

- ^：用来匹配字符串的开始。
- \$：用来匹配字符串的结尾。
- []：在方括号中的任何字符都可以匹配。例如[abc]表示匹配方括号中的字符a、字符b或者是字符c。
- -：连字符用来表示字符匹配的范围。例如[a-z]表示匹配方括号中字符a到字符z中的任何一个字符。
- +：表示用于匹配的该字符在被匹配的字符串中出现至少一次或者多次。
- \*：表示用于匹配的该字符在被匹配的字符串中出现零次或者多次。
- ()：在圆括号中的内容将被看做一个整体。例如(abc)，表示匹配样式(abc)的字符串是abc。
- {m}：其中整数m表示花括号前的字符串需要出现的次数。例如{abc}{2}表示匹配样式{abc}{2}的字符串是abccabc。

下面来看一个使用REGEXP关键字的例子。

例6.17 查询学生信息表中姓王的所有学生的信息（使用REGEXP关键字）。

```
SELECT stuID,stuName,age,sex,birth
FROM T_student
WHERE stuName REGEXP '^王'
```

这里使用的是REGEXP关键字查询学生信息表中姓王的所有学生的信息。使用REGEXP关键字取代例6.15中的LIKE关键字。在REGEXP关键字后使用“^”符号表示匹配的表示学生姓名的字符串是以“王”开头的。其查询结果如图6.16所示。

stuID	stuName	age	sex	birth
1115263	王海	23	男	1996-09-02
1221256	王嘉麟	24	男	1985-05-18
1231456	王玉梅	22	女	1987-03-28

图6.16 查询学生信息表中姓王的所有学生的信息（使用REGEXP关键字）

这条SQL语句是从学生信息表中选择所有以“王”字开头的学生信息。这里的WHERE子句中的REGEXP '^王'表示会匹配任何一个以“王”字开头的所有学生的信息。无论这个姓王的学生们的名字是两个字还是三个字，都会被检索出来。其查询结果与例6.15中的查询结果相同。

使用REGEXP关键字还可以对字符串进行模式匹配的测试，测试子串是否与被匹配的字符串匹配。语法格式如下：

```
string REGEXP pattern_string
```

该关键字的功能是如果测试的子串与被匹配的字符串匹配，则返回值为1，如果测试的子串与被匹配的字符串不匹配，则返回的值为0。其中，参数string是表示用来与匹配模板进行匹配的测试子串；参数pattern\_string表示被匹配的字符串。下面来看一个使用REGEXP关键字对字符串进行模式匹配测试的例子。

例6.18 对字符串进行模式匹配测试。

```
SELECT 'agf' REGEXP '[a-d]','banana' REGEXP '(ana){2}'
```

在这段SELECT语句中，一共对两个字符串的模式匹配进行了测试。第一个用来测试字符串agf是否与模式[a-d]+进行匹配，第二个用来测试字符串banana是否与模式(ana){2}进行匹配。其查询结果如



下所示。

+-----+-----+-----+-----+	
'agf' REGEXP '[a-d]+'	'banana' REGEXP '(ana){2}'
+-----+-----+-----+-----+	
1	0

从查询结果看，第一个模式匹配测试返回的值为1，第二个模式匹配测试返回的值为0。即第一个模式匹配的测试返回的结果为TRUE，第二个模式匹配测试返回的结果为FALSE。读者可以考虑一下这两个模式匹配的测试结果是如何产生的。

**提示** 模式[a-d]+表示匹配方括号中的任何一个字符一次或者多次，模式(ana){2}表示匹配字符串ana应该出现两次。

在MySQL数据库中，也可以使用NOT REGEXP关键字进行模式匹配，其用法与REGEXP关键字的用法相同，意义和REGEXP关键字相反。即如果测试的子串与被匹配的字符串匹配，则返回值为0，如果测试的子串与被匹配的字符串不匹配，则返回的值为1。

## 6.6 小结

本章主要介绍了在WHERE子句中的几种操作符的使用方法，包括比较运算符、逻辑运算符、空值查询以及模糊查询的使用。在使用比较运算符比较字符值时，比较运算符右侧的字符需要使用单引号，而且字符值在使用时需要区分大小写。在与NULL值进行比较时，不能使用比较运算符，而只能使用IS NULL和IS NOT NULL。另外在使用多个逻辑运算符进行查询操作时，在SQL语言中算术比较运算符的优先级高于逻辑运算符的优先级。使用算术运算符和逻辑运算符进行复杂查询操作时，可以使用括号处理和改变优先级。

## 第7章 表中数据的排序与分组

在前面介绍的使用SQL语句执行查询操作时，读者可能发现查询出的数据结果的排序是无序的。为了更好地观察数据表中的查询结果，开发人员或者用户经常要对查询的数据进行排序操作，这就需要使⽤ORDER BY子句。在数据库的实际应用中，有时需要对查询的数据进行统计和分组操作，这就需要了解SQL语句的聚合函数和GROUP BY子句的使用。本章将主要对这两部分的内容进行介绍。有些时候开发人员或者用户还希望对分组后的结果做进一步的统计，在SQL语句中提供了ROLLUP关键字用来对数据进行统计。在本章的最后还将介绍主要数据库中如何限制结果集的行数。

本章重点：

- ☐ 指定表中的一列进行排序
- ☐ 指定表中列的位置序号进行排序
- ☐ 对SELECT语句中的非选择列进行排序
- ☐ 指定表中的多列进行排序
- ☐ 常用的聚合函数
- ☐ 单列分组与多列分组
- ☐ 使用HAVING限制分组后的查询结果
- ☐ 对分组结果进行排序
- ☐ 使用ROLLUP关键字统计数据
- ☐ 不同数据库中限制结果集行数的方法

### 7.1 使用ORDER BY 子句对数据记录进行排序

如果想在SELECT语句中对所查询的结果按照某种顺序进行排序操作，那么就需要使用ORDER BY子句。使用ORDER BY子句可以对数据表中指定的某一列进行排序，也可以对数据表中指定的多个列进行排序操作。这一节将介绍使用ORDER BY子句实现对数据记录排序的方法。

#### 7.1.1 指定表中的一列进行排序

通过ORDER BY子句可以对查询结果中指定的列进行升序或者是降序操作，这取决于ORDER BY子句后的关键字，如果ORDER BY子句后面的关键字是ASC，则对查询的结果执行升序操作；如果ORDER BY子句后面的关键字是DESC，则对查询的结果执行降序操作。其语法规则如下：

```
ORDER BY 列名1 [ASC|DESC]
```

其中列名1表示需要对该列进行排序操作。关键字ASC和DESC是可选的。如果ORDER BY子句后面不写ASC或者是DESC，则默认执行的是升序操作。

例7.1 查询教师信息表中教师的工资，并按照教师工资从低到高排序。

```
SELECT teaID,teaName,dept,profession,salary
FROM T_teacher
ORDER BY salary ASC
```

这里ORDER BY子句用于对查询出来的教师工资salary结果进行排序。其中ASC关键字表示对指定的列进行升序排序。其查询结果如图7.1所示。

从查询结果可以看到，在图7.1中教师的排序是按照其工资收入从低到高的顺序显示的。当然，ORDER BY子句也可以将指定列进行降序排序。

例7.2 查询教师信息表中教师的工资，并按照教师工资从高到低排序。

```
SELECT teaID,teaName,dept,profession,salary
FROM T_teacher
ORDER BY salary DESC
```

这里的ORDER BY子句用于对查询出来的教师的工资salary进行降序排序，其中DESC关键字表示对指定的列进行降序排序。其查询结果如图7.2所示。

teaID	teaName	dept	profession	salary
t156354	王新	数学系	讲师	2500
t105320	于波	计算机系	讲师	2800
t102225	赵伟	计算机系	副教授	3000
t186585	孙立	物理系	讲师	3200
t181585	李慧	物理系	教授	3500
t103265	张晶	计算机系	教授	3800
t106358	毛翠	计算机系	教授	4000
t156355	李中	数学系	教授	4200

图7.1 查询教师信息表中教师的工资，  
并按照教师工资从低到高排序

teaID	teaName	dept	profession	salary
t156355	李中	数学系	教授	4200
t106358	毛翠	计算机系	教授	4000
t103265	张晶	计算机系	教授	3800
t181585	李慧	物理系	教授	3500
t186585	孙立	物理系	讲师	3200
t102225	赵伟	计算机系	副教授	3000
t105320	于波	计算机系	讲师	2800
t156354	王新	数学系	讲师	2500

图7.2 查询教师信息表中教师的工资，  
并按照教师工资从高到低排序

从查询结果可以看到，在图7.2的教师工资的排序与图7.1的教师工资的排序方式正好相反，是按照从高到低的顺序显示的。

这里需要说明的一点是，ORDER BY子句默认的排序方式是升序排序的。也就是说，在使用ORDER BY子句时，如果没有使用关键字ASC（升序）或者是DESC（降序）指定列的排序方式，则最终显示的结果会按照指定列的升序的方式来显示。读者可以把例7.1中SQL语句中的ORDER BY子句指定列后面的关键字ASC去掉，再查询一次，看一下查询的结果是否有变化。

**注意** 对于指定要排序的列，如果该列中存在空值的行，则在升序排序中，该行会显示在最前面；在降序排序中，该行会显示在最后面。

ORDER BY子句也可以放到WHERE子句之后，对满足条件的记录结果进行排序操作。例如下面这个例子。

例7.3 查询教师信息表中计算机系的教师的工资，并按照教师工资从低到高排序。

```
SELECT teaID,teaName,dept,profession,salary
FROM T_teacher
WHERE dept = '计算机系'
ORDER BY salary ASC
```

使用WHERE子句限制查询的结果，在WHERE子句之后，使用ORDER BY子句对计算机系的教师

## 零基础学SQL

工资进行升序排序。其查询结果如图7.3所示。

注意

ORDER BY 子句只是对最终结果进行排序，不能用在子查询的SELECT语句中，而且ORDER BY子句的位置必须放在所有子句的最后，也就是说，如果SELECT子句中有多个子句（如WHERE子句、GROUP BY子句、HAVING子句等），那么ORDER BY子句必须在这些子句之后，作为最后一条子句出现。

teaID	teaName	dept	profession	salary
t105320	于波	计算机系	讲师	2800
t102225	赵伟	计算机系	副教授	3000
t103265	张磊	计算机系	教授	3800
t106358	毛翠	计算机系	教授	4000

图7.3 查询教师信息表中计算机系的教师的工资，并按照教师工资从低到高排序

### 7.1.2 指定表中列的位置序号进行排序

在使用ORDER BY子句进行排序操作时，除了可以使用列名对指定列进行排序，也可以使用该列在选择列表中的位置的序号对其进行排序。这里还以教师信息表中教师工资排序为例，看一下如何使用列的位置序号对教师工资进行排序。

例7.4 查询教师信息表中教师的工资，并按照教师工资从低到高排序（使用列序号方式）。

```
SELECT teaID,teaName,dept,profession,salary
FROM T_teacher
ORDER BY 5 ASC
```

这里的查询语句与例7.1中的SQL语句基本相同，只是在最后的ORDER BY子句中，将表示教师工资所在列的列名salary用该列在选择列表中的位置的序号来代替。因此在ORDER BY子句中用序号5来替代列名salary，其查询结果如图7.4所示。

teaID	teaName	dept	profession	salary
t156354	王莉	数学系	讲师	2500
t105320	于波	计算机系	讲师	2800
t102225	赵伟	计算机系	副教授	3000
t186595	孙立	物理系	讲师	3200
t181595	李慧	物理系	教授	3500
t103265	张磊	计算机系	教授	3800
t106358	毛翠	计算机系	教授	4000
t156365	李中	数学系	教授	4200

图7.4 查询教师信息表中教师的工资，并按照教师工资从低到高排序（使用列序号方式）

从查询结果可以看到，图7.4的查询结果和图7.1的查询结果是完全一样的。但是使用列在选择列表中的位置的序号来代替列名进行排序的方法在有些场合下却是很有用的，在ORDER BY子句中使用列序号进行排序操作，一般用于下面两个场合。

- ❑ 如果一个列的列名的字符很多，在ORDER BY子句中就可以使用列位置序号的方式简化排序语句的长度。
- ❑ 在ORDER BY子句中使用列序号方式进行排序操作的另外一个用处就是在使用UNION、MINUS等这些集合操作的时候，此时如果需要对数据表中的选择列进行排序，就可以使用列序号的方式。

### 7.1.3 对SELECT语句中的非选择列进行排序

在上面的这些例子中，使用ORDER BY子句排序的列都出现在SELECT语句的查询列表中。但是ORDER BY子句中也可以对没有在选择语句中出现的列进行排序操作。看看下面这个例子。

例7.5 查询教师信息表中计算机系教师的工资，并按照工资从低到高排序（表示工资的列不出现）。

```
SELECT teaID,teaName,dept,profession
FROM T_teacher
WHERE dept = '计算机系'
```

ORDER BY salary

在这个查询语句中，SELECT语句中没有出现表示教师工资的列，ORDER BY子句中对教师工资列进行升序排序（ORDER BY子句后面没有ASC关键字，默认情况下是升序排序）。其查询结果如图7.5所示。

将图7.5与图7.3比较可以发现，其显示出来的教师的排序是一样的，只不过图7.5中没有显示教师工资列。

teaID	teaName	dept	profession
t105320	于波	计算机系	讲师
t102225	赵伟	计算机系	副教授
t103265	张晶	计算机系	教授
t106358	毛翠	计算机系	教授

图7.5 按照计算机系教师的工资从低到高排序（表示工资的列不出现）

**说明** 通常情况下，使用ORDER BY子句对列表进行排序时，都会使用在SELECT语句中出现的选择列表的列进行排序操作，以便可以更直观地观察数据。

7.1.4 指定表中的多列进行排序

ORDER BY子句除了可以指定单列进行排序操作，也可以指定数据表中的多个列进行排序操作。如果要指定数据表中的多个列进行排序操作，则指定排序的列与列之间需要用逗号隔开。其语法规则如下：

ORDER BY 列名1,列名2 [ASC|DESC]

其中列名1和列名2表示需要对指定的数据列进行排序操作。列名1和列名2之间用逗号进行分隔。关键字ASC和DESC是可选的。如果ORDER BY子句后面不写ASC或者是DESC，则默认执行的是升序操作。下面来看一个指定表中的多列进行排序的例子。

**例7.6** 在教师信息表中，按照教师编号和教师工资的升序方式对其进行排序。

```
SELECT teaID,teaName,dept,profession,salary
FROM T_teacher
ORDER BY dept,salary
```

其查询结果如图7.6所示。

从这个查询结果可以看到，在最终显示的结果中，教师的编号是按照升序排列的，而教师工资却并没有完全按照升序的顺序排序，这是什么原因呢？要了解这样的结果产生的原因，就需要知道ORDER BY子句指定表中的多个列进行排序的排序规则。使用ORDER BY子句指定表中的多个列进行排序时，其排序的方式遵循以下规则：

- (1) 根据ORDER BY子句中指定的第一列，按照指定的升序或者是降序方式进行排序。
- (2) 当ORDER BY子句中指定的第一列中出现相同数据时，再根据ORDER BY子句中指定的第二列的升序或者降序方式进行排序。
- (3) 当ORDER BY子句中指定的第二列中出现相同数据时，会根据ORDER BY子句中指定的第三列的升序或者降序方式进行排序，依次类推。

根据这个规则，也就不难理解上述结果产生的原因了。当然在使用ORDER BY子句指定表中的多个列进行排序时，指定列的排序方式也可以不同。为了更好地理解ORDER BY子句中指定表中的多个

teaID	teaName	dept	profession	salary
t105320	于波	计算机系	讲师	2600
t102225	赵伟	计算机系	副教授	3000
t103265	张晶	计算机系	教授	3600
t106358	毛翠	计算机系	教授	4000
t156354	王新	数学系	讲师	2500
t156355	李中	数学系	教授	4200
t186595	孙立	物理系	讲师	3200
t181585	李慧	物理系	教授	3500

图7.6 按照教师信息表中教师编号和教师工资的升序方式对其进行排序



## 零基础学SQL

列进行排序的排序规则，下面再来看两个例子。

**例7.7** 在教师信息表中，按照教师所在院系编号升序和教师工资的降序方式对其进行排序。

```
SELECT teaID,teaName,dept,profession,salary
FROM T_teacher
ORDER BY dept ASC,salary DESC
```

这里指定教师所在院系编号和教师工资两个列进行排序操作。其中教师所在院系编号按照升序的方式排序，这里使用ASC关键字（也可以不写ASC关键字）；教师工资按照降序的方式排序，这里使用DESC关键字。其查询结果如图7.7所示。

如果将例7.7中指定的教师所在院系编号的列和教师工资的列的排序顺序颠倒一下，其最终的查询结果会有什么变化呢？

**例7.8** 在教师信息表中，按照教师教师工资的降序和所在院校编号升序方式对其进行排序。

```
SELECT teaID,teaName,dept,profession,salary
FROM T_teacher
ORDER BY salary DESC,dept ASC
```

这里也是指定教师所在院校编号和教师工资两个列进行排序操作。其中，教师工资按照降序的方式排序，这里使用DESC关键字；教师所在院校编号按照升序的方式排序，这里使用ASC关键字（也可以不写ASC关键字）。其查询结果如图7.8所示。

可以看到，图7.8的查询结果和图7.7的查询结果所显示的内容的顺序是完全不同的。通过上面两个例子的查询结果，读者也可以体会一下使用ORDER BY子句指定表中的多个列进行排序的排序规则。

teaID	teaName	dept	profession	salary
t106358	毛翠	计算机系	教授	4000
t103265	张晶	计算机系	教授	3800
t102225	赵伟	计算机系	副教授	3000
t105320	于波	计算机系	讲师	2800
t156355	李中	数学系	教授	4200
t156354	王新	数学系	讲师	2500
t181585	李慧	物理系	教授	3500
t186585	孙立	物理系	讲师	3200

图7.7 按照教师信息表中教师所在院系编号升序和教师工资的降序方式对其进行排序

teaID	teaName	dept	profession	salary
t156355	李中	数学系	教授	4200
t106358	毛翠	计算机系	教授	4000
t103265	张晶	计算机系	教授	3800
t181585	李慧	物理系	教授	3500
t186585	孙立	物理系	讲师	3200
t102225	赵伟	计算机系	副教授	3000
t105320	于波	计算机系	讲师	2800
t156354	王新	数学系	讲师	2500

图7.8 按照教师信息表中教师工资的降序和所在院系编号升序方式对其进行排序

**提示** 在使用ORDER BY子句指定表中的多个列进行排序时，首先基于ORDER BY子句中指定的第一个列进行排序，在第一列的值相同的情况下，再根据ORDER BY子句中指定的第二个列进行排序，并按照这种方式依次类推。

## 7.2 常用的聚合函数

聚合函数也被称为分组函数或者统计函数，主要用于对得到的一组数据进行统计计算，例如求和、求平均值等，常用的聚合函数包括COUNT、MAX、MIN、SUM和AVG五个。表7.1中列出了这5个函数及其功能。

□ COUNT、SUM和AVG函数中可以使用DISTINCT关键字去除指定列中的重复项。使用DISTINCT关键字后只是对不同行的值进行统计。

□ MAX和MIN函数中的列或者表达式可以是数字型、字符型或者是日期类型的值。如果MAX和



- MIN函数中的列或者表达式是字符型的，则按照首字母从A到Z的顺序排序，如果首字母相同，则比较字符串中第二个字母的大小，依次类推。汉字则是按照其汉语拼音的全拼来排序。
- ❑ SUM和AVG函数中的表达式只能是数字类型的值。
  - ❑ 除了COUNT(\*)之外，其他的几个函数在计算时都忽略表达式中的空值（NULL行）。
  - ❑ COUNT函数是用来计算数据表中的总行数，SUM函数是用来计算数据表中某一列的属性值的总和（可以参看例7.13）。

表7.1 聚合函数及其功能

函 数	功 能
COUNT(列表达式)	计算给定列或者表达式中非空行的行数
COUNT(*)	计算数据表中的总行数，包括空值（NULL行）
MAX(列表达式)	计算给定列或者表达式中的最大值
MIN(列表达式)	计算给定列或者表达式中的最小值
SUM(列表达式)	计算给定列或者表达式中所有值的总和
AVG(列表达式)	计算给定列或者表达式中所有值的平均值

下面通过几个例子来看一下如何在SQL语句中使用这些聚合函数。

例7.9 查询教师信息表中教师的数量。

```
SELECT COUNT(*)
FROM T_teacher
```

其查询结果如图7.9所示。

例7.10 查询教师信息表中院系的数量。

```
SELECT COUNT(dept)
FROM T_teacher
```

其查询结果如图7.10所示。

例7.11 查询教师信息表中不重复的院系的数量。

```
SELECT COUNT( DISTINCT dept) AS 院系数量
FROM T_teacher
```

其查询结果如图7.11所示。

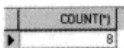


图7.9 查询教师信息表中教师的数量

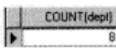


图7.10 查询教师信息表中院系的数量



图7.11 查询教师信息表中不重复的院系的数量

从这个查询结果可以看到，当使用DISTINCT关键字之后，图7.11所显示的数量为3，而没使用DISTINCT关键字查询出来的结果为8，通过使用DISTINCT关键字将在教师信息表中重复的院系全都去掉了。

例7.12 查询教师信息表中的最高工资和最低工资。

```
SELECT MAX(salary),MIN(salary)
FROM T_teacher
```

## 零基础学SQL

其查询结果如图7.12所示。

**例7.13** 查询教师信息表中的教师的工资总和以及教师的平均工资。

```
SELECT SUM(salary),COUNT(salary),AVG(salary)
FROM T_teacher
```

其查询结果如图7.13所示。

MAX(salary)	MIN(salary)
4200	2500

图7.12 查询教师信息表中的最高工资和最低工资

SUM(salary)	COUNT(salary)	AVG(salary)
27000	8	3375.0000

图7.13 查询教师信息表中的教师的工资总和以及教师的平均工资

从例7.13的查询结果中，读者可以了解到SUM函数、COUNT函数和AVG函数之间的区别和联系。

- ❑ SUM函数计算的是教师信息表中列salary中全体教师工资的总和，COUNT函数是计算教师信息表中共有几条教师记录。
- ❑ AVG函数是用来计算教师信息表中教师的平均工资，其计算结果与SUM(salary)/COUNT(salary)的计算结果相同。

### 说明

聚合函数一般与GROUP BY子句在一起使用。GROUP BY子句是用于对查询结果进行分组。有关GROUP BY子句内容将在后面的几个小节中做详细的介绍。

聚合函数只能出现在SELECT语句、GROUP BY子句以及HAVING子句中，WHERE子句中不能出现聚合函数。

## 7.3 使用GROUP BY子句对表中数据进行分组

GROUP BY子句可以根据给定数据列的多个数据查询出来的结果进行分组，它既可以对单列数据进行分组，也可以对多列数据进行分组。在GROUP BY子句后还可以使用HAVING子句对分组后的结果做进一步的筛选。这一节就来介绍使用GROUP BY子句对表中数据进行分组的方法。

### 7.3.1 单列分组

使用GROUP BY子句对数据表中的某一列进行分组时，会对指定分组的列中不同的值都计算出一个统计结果。其语法格式如下：

```
GROUP BY 列名1
```

其中列名1表示需要对该列进行分组操作。下面来看一个使用GROUP BY子句对数据表中的单个列进行分组的例子。

**例7.14** 对教师信息表中的院校进行分组，并统计出每个院校中拥有教师职称的数量。

```
SELECT dept,COUNT(profession)
FROM T_teacher
GROUP BY dept
```

其查询结果如图7.14所示。

dept	COUNT(profession)
计算机系	4
数学系	2
物理系	2

图7.14 对教师信息表中的院校进行分组，并统计出每个院校中拥有教师职称的数量

这条SQL语句会在教师信息表（T\_teacher）中将dept字段中不同院校的值都挑选出来。在T\_teacher表中一共有3个院系，分别是计算机系、数学系和物理系，首先是将这3个院系选取出来，并将其分成3组，然后再将每一个院系中拥有职称的教师数量都统计出来。这里的教师职称包括3类：教授、副教授和讲师。从查询的结果可以看到，计算机系中拥有教师职称的教师数量是4，数学系中拥有教师职称的教师数量是2，物理系中拥有教师职称的教师数量是2。

### 7.3.2 多列分组

使用GROUP BY子句对数据表中的多个列进行分组时，会对指定分组的多个列中不同的值都计算出一个统计结果。其语法格式如下：

GROUP BY 列名1, 列名2...

其中列名1和列名2表示需要对指定列进行分组操作。列名1和列名2之间用逗号进行分隔。下面来看一个使用GROUP BY子句对数据表中的多个列进行分组的例子。

**例7.15** 对教师信息表中的院系和教师职称进行分组，并统计出该院系教师的最高工资。

```
SELECT dept,profession,MAX(salary)
FROM T_teacher
GROUP BY dept,profession
```

其查询结果如图7.15所示。

这条SQL语句是对教师信息表中的表示院系的列和表示教师职称的列进行分组。教师信息表中的院系有三个，分别是计算机系、数学系和物理系。每一个院系中对应的教师职称有三种，分别是教授、副教授和讲师。对表示院系的列和表示教师职称的列进行分组后，最多可能会产生9种结果。在例7.15的查询语句中共查询出7条记录。因为在教师信息表（T\_teacher）里，数学系和物理系中没有职称为副教授的教师信息，读者可以在T\_teacher表中添加两条相关的信息，看一下查询结果会有什么样的变化。

读者可能注意到，在例7.14和例7.15中，SELECT语句中同时包含有数据列和聚合函数。这里需要说明的是，如果SELECT语句中同时包含有数据列和聚合函数，那么必须使用GROUP BY子句进行分组操作。如果在SELECT语句中同时包含有数据列和聚合函数，不使用GROUP BY子句，在执行SQL语句时就会出现错误。例如下面的SQL语句是错误的。

```
SELECT dept,profession,MAX(salary)
FROM T_teacher
```

在这条SQL语句中，SELECT语句中含有MAX函数，但是并没有使用GROUP BY子句对其进行分组。在MySQL 5.0中执行这样一个SQL语句，会得到如下一条错误信息。

```
Mixing of GROUP columns (MIN(),MAX(),COUNT(),...) with no GROUP columns is illegal
if there is no GROUP BY clause
```

这条错误信息是说，在使用MIN()、MAX()、COUNT()等这样的聚合函数时，需要使用GROUP BY子句，如果没有GROUP BY子句，这条SQL语句是非法的。

dept	profession	MAX(salary)
计算机系	副教授	3000
计算机系	讲师	2800
计算机系	教授	4000
数学系	讲师	2500
数学系	教授	4200
物理系	讲师	3200
物理系	教授	3900

图7.15 对教师信息表中的院系和教师职称进行分组，并统计出该院系教师的最高工资

**注意** SELECT语句中都同时包含有数据列和聚合函数，使用GROUP BY子句进行分组时，在GROUP BY子句中包括了SELECT语句中出现的所有选择列。否则分组的信息就可能出现错误。

### 7.3.3 使用HAVING限制分组后的查询结果

如果想要对分组后的结果限制查询条件，就需要使用HAVING子句。由于HAVING子句是用来限制分组后的查询结果，所以该子句需要放到GROUP BY子句的后面使用。其语法格式如下：

GROUP BY 列名1 HAVING 条件表达式

其中列名1表示需要对该列进行分组操作。HAVING子句后的条件表达式是用来筛选分组后的结果。在HAVING子句中经常使用聚合函数对分组后的结果进行筛选。

**例7.16** 对教师信息表中的院系和教师职称进行分组，并统计出该院系教师最高工资>3000元的记录。

```
SELECT dept,profession,MAX(salary)
FROM T_teacher
GROUP BY dept,profession
HAVING MAX(salary)>3000
```

这条SQL语句是在使用GROUP BY子句对院系和教师职称进行分组后，通过HAVING子句在分组后的结果中进一步筛选出教师最高工资>3000元的教师信息。其查询结果如图7.16所示。

dept	profession	MAX(salary)
计算机系	教授	4000
数学系	教授	4200
物理系	讲师	3200
物理系	教授	3500

图7.16 院系和教师职称进行分组后，并统计出该院校教师最高工资>3000元的记录

HAVING子句是用来对分组后的查询结果进行筛选。也就是说，HAVING子句是针对GROUP BY子句的，它作用于组。而WHERE子句是在分组之前对满足条件的元组进行查询，它是针对于SELECT语句的，是作用在表或者视图上。下面来看一个WHERE子句和HAVING子句混合使用的例子。

**例7.17** 对教师信息表中的职称进行分组，并统计出年龄>30教师最高工资不低于3000元的记录。

```
SELECT profession,MAX(salary)
FROM T_teacher
WHERE age>30
GROUP BY profession
HAVING MAX(salary)>=3000
```

其中，WHERE子句用来限制教师信息表中的教师的年龄，HAVING子句在GROUP BY子句之后，用来对教师职称分组后的查询结果进行限制，限制条件是教师最高工资> 3000元，也就是筛选出教师最高工资不低于3000元的教师记录。其查询结果如图7.17所示。

profession	MAX(salary)
副教授	3000
讲师	3200
教授	4200

图7.17 查询对职称分组后年龄>30教师最高工资不低于3000元的记录

这条SQL语句中，首先会执行WHERE子句中的限制条件，将年龄大于30的教师记录查询出来，而HAVING子句会在WHERE子句之后执行，HAVING子句的工作是对分组后的查询结果做进一步的限制。

**注意** 在HAVING子句可以使用聚合函数统计有关字段的计算结果；在WHERE子句中不允许使用聚合函数。

HAVING子句中不仅可以出现在SELECT语句中出现的列，也可以使用SELECT语句中没有出现的列。来看下面这个例子。

**例7.18** 对教师信息表中的院系和教师职称进行分组，并显示出教师最高工资>3000元的教师姓名。

```
SELECT teaName,dept,profession
FROM T_teacher
GROUP BY dept,profession
HAVING MAX(salary)>3000
```

在这条SQL语句中，SELECT语句里并没有出现查询教师工资的列，在HAVING子句中使用MAX函数对分组后的教师最高工资进行筛选，筛选出教师最高工资>3000元的教师信息。其查询结果如图7.18所示。

**说明** 在GROUP BY子句的后面使用HAVING子句时，最终返回的只会是满足HAVING子句筛选后的结果。

teaName	dept	profession
张晶	计算机系	教授
李中	数学系	教授
孙立	物理系	讲师
李慧	物理系	教授

图7.18 显示对院系和教师职称分组后教师最高工资>3000元的教师姓名

### 7.3.4 对分组结果进行排序

很多时候，对数据表中数据进行分组后，还希望对分组的结果进行排序操作。如果想对使用了GROUP BY子句的分组结果进行排序，就需要使用ORDER BY子句。

**例7.19** 对教师信息表中的院系进行分组，并根据每个院系中教师的最高工资降序排序。

```
SELECT dept,profession,MAX(salary)
FROM T_teacher
GROUP BY dept
ORDER BY MAX(salary) DESC
```

其查询结果如图7.19所示。

**注意** GROUP BY子句和ORDER BY子句同时使用时，ORDER BY子句要放到最后，它只是对最终的查询结果进行排序。

dept	profession	MAX(salary)
数学系	讲师	4200
计算机系	副教授	4000
物理系	教授	3500

图7.19 对教师信息表中的院系进行分组，并根据每个院系中教师的最高工资降序排序

### 7.3.5 GROUP BY子句中处理NULL值

在使用GROUP BY子句对指定列进行分组时，有时可能会遇到指定列中含有NULL值的情况。此时，GROUP BY子句会将该列中所有的NULL值归为一组。

**例7.20** 对教师信息表中的教师津贴字段进行分组。

```
SELECT COUNT(*),pension
FROM T_teacher
GROUP BY pension
ORDER BY pension ASC
```

COUNT(*)	pension
3	260.5
1	278
1	289.3
1	300
1	310.2

图7.20 对教师信息表中的教师津贴字段进行分组

这段SQL语句是对教师信息表中的教师津贴字段进行分组，并统计出分组结果，按照查询出的教师津贴的值从小到大升序排序。其查询结果如图7.20所示。

从查询结果中可以看到，在对教师信息表中表示教师津贴的列pension进行分组时，将列pension中



含有空值的记录都归为1组，在教师信息表中，教师津贴的列pension为NULL的记录一共有3条。这3条记录在显示的结果中被归为一组并显示在分组结果的最前面。

## 7.4 使用ROLLUP关键字统计数据

前面一节中已经介绍了使用GROUP BY子句可以对数据表中的一列或者多列进行分组，查询数据统计的结果。在实际应用中，有时不仅需要得到分组后的统计结果，还希望对分组的统计结果做进一步的计算，例如通过对教师信息表（T\_teacher）中的院系和教师职称进行分组，得到分组后教师的最高工资，得到这个最高工资后，还希望对每一个院系中的教师的最高工资做一个阶段性的统计，希望得到各个院系中不同职称的教师的最高工资的加和（相当于小计），还希望得到所有院系不同职称教师最高工资的总和（相当于总计）。这个时候仅仅使用GROUP BY子句是无法做到的，此时就需要使用ROLLUP关键字。

ROLLUP关键字的作用是对分组后的数据做阶段性的操作（相当于小计），并且可以对查询出来的所有记录的数据做一个最终操作（相当于总计）。小计和总计的运算结果会自动添加到分组后的数据列中以及查询结果的最后一行中。ROLLUP关键字使用时需要放到GROUP BY关键字的后面。ROLLUP关键字在不同的数据库中的使用方式上稍有不同。

### 1. 在MySQL 5.0和Microsoft SQL Server数据库中使用ROLLUP关键字统计数据

在MySQL 5.0和Microsoft SQL Server数据库中需要使用WITH ROLLUP。其语法格式如下：

```
GROUP BY 列名1 WITH ROLLUP
```

其中列名1表示要对该列进行分组，WITH ROLLUP关键字表示要对分组的结果进行统计。当然也可以对多个列进行分组，并统计分组后的结果。其语法格式如下：

```
GROUP BY 列名1 ,列名2 WITH ROLLUP
```

对多个列进行分组时，列名与列名之间需要用逗号分隔。下面来看一个在MySQL 5.0数据库中使用ROLLUP关键字进行统计数据的例子。

例7.21 对教师信息表中的院系和教师职称进行分组，并对分组后的教师工资进行统计。

```
SELECT dept,profession,SUM (salary)
FROM T_teacher
GROUP BY dept,profession WITH ROLLUP
```

在MySQL 5.0和Microsoft SQL Server数据库中，ROLLUP关键字前还需要有一个WITH关键字，而且WITH ROLLUP要放到GROUP BY子句的后面。其查询结果如图7.21所示。

从查询结果可以看到，计算机在对教师信息表中的院系和教师职称进行分组过程中，在每一个分组完成之后都自动添加了一条统计记录。例如在完成对计算机系和教师职称的分组之后，在显示数据表的第4行自动多出了一条统计记录，在这条记录中统计出了计算机系不同职称的教师的工资总和，这里可以看到计算机系不同职称教师的工资总和是13600元。同样，在完成对

dept	profession	sum(salary)
计算机系	副教授	3000
计算机系	讲师	2800
计算机系	教授	7800
计算机系	总计	13600
数学系	讲师	2500
数学系	教授	4200
数学系	总计	6700
物理系	讲师	3200
物理系	教授	3500
物理系	总计	6700
总计	总计	27000

图7.21 对教师信息表中的院系和教师职称分组后的教师工资进行统计



数学系和教师职称的分组以及物理系和教师职称的分组之后。在显示数据表的第7行和第10行都多出了一条统计记录，分别计算数学系和物理系不同职称教师的工资总和。这里数学系和物理系教师工资统计后的结果都是6700元。

另外，在数据表的最后一行，还多出了一条统计记录，这条记录就是对数据表中的所有记录进行统计，选择所有记录中教师工资的总和进行加和操作，并将该统计结果显示在数据表的最后一行。这里最后计算出来的结果是27000元。

## 2. 在Oracle数据库中使用ROLLUP关键字统计数据

在Oracle数据库中，ROLLUP关键字需要紧跟在GROUP BY关键字的后面，然后再写需要分组的字段。其语法格式如下：

GROUP BY ROLLUP (列名1,列名2...)

其中列名1与列名2表示要对指定列进行分组，ROLLUP关键字表示要对分组的结果进行统计。GROUP BY子句后面用括号括起来的列名可以是一列，也可以是多列，如果是多个列，列名与列名之间需要用逗号分隔。下面仍以例7.21为例，来看一下在Oracle数据库中如何使用ROLLUP关键字对数据进行统计。其SQL语句如下：

```
SELECT dept,profession, SUM (salary)
FROM T_teacher
GROUP BY ROLLUP (dept,profession)
      dept      profession      MAX(salary)
-----
计算机系      副教授      3000
计算机系      讲师      2800
计算机系      教授      7800
计算机系      13600
数学系      讲师      2500
数学系      教授      4200
数学系      6700
物理系      讲师      3200
物理系      教授      3500
物理系      6700
物理系      27000
```

这里用粗体字突出显示的就是ROLLUP关键字进行加和后的统计结果。其查询结果和图7.21中显示的结果是一样的，读者可以在SQL\*Plus中运行这段SQL语句，看一下运行结果。

### 注意

在SQL\*Plus中运行SQL语句时，需要在SQL语句的结尾加上一个分号。否则SQL语句可能会无法正常运行。

## 7.5 限制结果集行数

有些时候，开发人员或者用户并不希望将查询结果的数据列中的数据全部显示出来，而是只希望显示其中的几行，尤其是在需要分页的操作中。例如，一个数据表最后查询出了100条记录，而开发人员或者用户只关心其中前10条记录的值，这就需要对查询结果中的数据记录的行数进行限制。在不同

零基础学SQL

的数据库中限制结果集行数的方法也不尽相同，这里就以MySQL数据库、Oracle数据库和Microsoft SQL Server数据库为例，介绍如何在这3种数据库中实现限制结果集行数的操作。

7.5.1 使用MySQL数据库限制结果集行数

在MySQL数据库中限制结果集行数可以使用LIMIT关键字，它可以用来限制查询出来的数据结果的个数。通过使用LIMIT关键字可以让开发人员或者用户得到其中想要的部分的结果。如果要使用LIMIT限制结果集行数，可以使用下面的语法格式。

```
LIMIT n
```

其中LIMIT是关键字，数字n表示要限制结果集行数。下面来看一个使用MySQL数据库限制结果集行数的例子。

例7.22 查询教师信息表中的教师信息，只显示按照教师编号升序排序后的前3条记录。

```
SELECT teaID,teaName,dept,profession
FROM T_teacher
ORDER BY teaID
LIMIT 3
```

这里的LIMIT 3表示查询教师信息表中的教师信息后，只显示查询出来的教师记录中的前3条记录。其查询结果如图7.22所示。

这里的查询结果中只显示了教师信息表中的前3条记录，后面的教师记录并没有显示出来。如果想查看教师信息表中后面的教师记录，可以使用下面的LIMIT形式来完成。其语法格式如下：

```
LIMIT offset,n
```

其中offset表示的是一个从数据表中的第1条记录开始算起的记录行的偏移值。起始偏移值是从0开始计算的，因此MySQL数据库中的操作实际上是从数据表中第offset +1条记录开始进行的。n表示记录行中要返回几条记录。其中offset和n都应该是数字值。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t103265	张磊	计算机系	教授
t105320	于洪	计算机系	讲师

图7.22 显示教师信息表中按照教师编号升序排序后的前3条记录

**说明** LIMIT 0,n和LIMIT n的查询结果是相同的。即LIMIT 0,n等价于LIMIT n。

例7.23 查询教师信息表中的教师信息，只显示按照教师编号升序排序后的第4条到第6条记录。

```
SELECT teaID,teaName,dept,profession
FROM T_teacher
ORDER BY teaID
LIMIT 3,3
```

这里的LIMIT 3,3表示的是从数据表中的第4条记录开始算起，再返回教师信息表中3条有关教师信息的记录。也就是要将教师信息表中第4条、第5条和第6条记录的值选取出来。其查询的结果如图7.23所示。

从图7.23显示的结果看，图7.23中按照教师编号升序的排序顺序又显示了3条记录。也就是显示了数据表中第4条、第5条和第6条记录的值。

teaID	teaName	dept	profession
t106350	毛翠	计算机系	教授
t156354	王新	数学系	讲师
t156355	李中	数学系	教授

图7.23 显示教师信息表中按照教师编号升序排序后的第4条到第6条记录

**注意** MySQL中数据记录起始偏移值是从0开始计算的，在MySQL数据库中，如果LIMIT关键字使用两个参数offset和n来限制查询的结果集行数，其操作实际上是从数据表中第offset+1条记录开始进行的。

### 7.5.2 使用Oracle数据库限制结果集行数

Oracle数据库中不支持类似于MySQL中的LIMIT关键字来限制结果集行数，但是在Oracle数据库中可以使用ROWNUM关键字限制结果集的行数。其语法格式如下：

```
WHERE ROWNUM<n
```

其中ROWNUM关键字表示对符合条件结果的序列号，它的起始值总是从1开始的。数字n表示要限制的结果集的行数。当然，这里的比较运算符除了可以使用小于(<)以外，还可以使用小于等于(<=)。例如，对于例7.22，使用ROWNUM关键字的SQL语句如下：

```
SELECT teaID,teaName,dept,profession
FROM T_teacher
WHERE ROWNUM<=3
ORDER BY teaID
```

这段SQL语句中ROWNUM关键字用来限制结果集的行数，这里让ROWNUM<=3，表示只希望返回其中的前3条教师记录。其查询结果如下所示。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t103265	张昌	计算机系	教授
t105320	于波	计算机系	讲师

如果希望返回教师信息表中第3条以后的教师记录，直接使用ROWNUM>3是错误的，因为ROWNUM起始值总是从1开始的。如果要查询教师信息表中第4条到第6条记录，也就是要查询ROWNUM在第4行到第6行之间的数据，需要通过子查询来完成。其SQL语句如下：

```
SELECT teaID,teaName,dept,profession
FROM
(SELECT ROWNUM AS rn,teaID,teaName,dept,profession
FROM
T_teacher
WHERE ROWNUM<=6)
where rn>=4
```

在这个SQL语句中，使用了一个子查询语句（有关子查询的内容将在第9章做详细的介绍），该子查询语句检索T\_teacher表，并在WHERE子句中使用ROWNUM关键字限制结果集的行数。其查询结果如下所示。

teaID	teaName	dept	profession
t106358	毛翠	计算机系	教授
t156354	王新	数学系	讲师
t156355	李中	数学系	教授



在SQL执行的过程中，首先通过一个子查询让它返回小于等于6的记录行，并为ROWNUM起一个列别名rn，然后在主查询中使用ROWNUM的列别名判断大于等于3的记录行。最后查询出来的结果就是返回教师信息表中的第4条到第6条教师记录。

**注意** 在数据库中含有大量数据的时候，使用这种操作方法会对查询速度有一定的影响。

### 7.5.3 使用Microsoft SQL Server数据库限制结果集行数

在MySQL数据库和Oracle数据库中使用LIMIT关键字和ROWNUM的方法限制结果集行数，在Microsoft SQL Server数据库中需要使用TOP关键字。其语法格式如下：

```
SELECT TOP n [PERCENT] 列名1,列名2 ...  
FROM 表名  
...
```

其中，TOP是表示限制结果集行数的关键字；数字n表示限制结果集行数；PERCENT关键字表示返回查询的结果集中前n%的行数，它是可选的；列名1与列名2表示要查询的数据表中列的名字，多个列名之间需要用逗号分隔。

例如，对于例7.22，使用TOP关键字限制结果集行数的SQL语句如下：

```
SELECT TOP 3 teaID,teaName,dept,profession  
FROM T_teacher
```

这段SQL语句中，TOP关键字用来限制结果集的行数，TOP关键字后面紧跟的数字3表示只希望返回其中的前3条教师记录。其查询结果如下所示。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t103265	张昌	计算机系	教授
t105320	于波	计算机系	讲师

除了可以指定行数限制查询结果中返回的结果集之外，也可以在TOP语句中使用PERCENT关键字返回指定百分比数量的行数。例如下面这个例子。

**例7.24** 返回教师信息表中前50%的行数。

```
SELECT TOP 50 PERCENT teaID,teaName,dept,profession  
FROM T_teacher
```

这段SQL语句中，TOP关键字用来限制结果集的行数，50 PERCENT指定返回结果集行数的百分比，表示返回的行数占教师信息表中总行数的50%。例7.9中已经查询出教师信息表中共有8条记录。那么TOP 50 PERCENT语句最终返回的查询结果应该是教师信息表中前4行的教师信息。其查询结果如下所示。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t103265	张昌	计算机系	教授
t105320	于波	计算机系	讲师
t106358	毛翠	计算机系	教授

**注意** TOP语句除了可以用在SELECT语句中，也可以用在INSERT、UPDATE和DELETE这样的数据更新语句中。

## 7.6 小结

本章主要介绍了使用ORDER BY子句对数据记录进行排序和使用GROUP BY子句对表中数据进行分组的方法。在使用ORDER BY子句对数据记录进行排序时，要注意ORDER BY子句放置的位置。在SQL语句中ORDER BY子句一定是作为其中的最后一条子句。另外需要了解使用ORDER BY子句对多个列进行排序的方法。

在使用GROUP BY子句对表中数据进行分组时，经常会使用聚合函数，需要掌握常用的5种聚合函数的使用方法。GROUP BY子句后可以跟一个HAVING子句对分组后的查询结果进行筛选。在WHERE子句中不能出现聚合函数。

另外，还应该了解ROLLUP关键字的作用以及在不同数据库中的使用方法。不同的数据库限制结果集行数使用的关键字也不相同。MySQL数据库中使用的是LIMIT关键字，Oracle数据库中使用的是ROWNUM关键字，而在Microsoft SQL Server数据库中使用的是TOP关键字。





## 第8章 连接查询与集合查询

在对数据库的查询过程中，有些时候检索一张表中的数据记录往往不能满足开发人员或者客户的需要。例如，查询学生的选课成绩信息。而学生选课信息和课程成绩信息分别在两个不同的数据表中。其中，在课程信息表（T\_curriculum）中包括课程的编号、课程的名字、课程的学分、课时以及授课的教师等学生选课信息，而学生的编号、选课的课程编号以及课程成绩等信息在成绩信息表（T\_result）中，此时为了在查询的结果中显示学生信息和所选课程的相关信息，就需要同时检索学生信息表（T\_student）和课程信息表（T\_curriculum），这就需要进行连接查询的操作。连接查询主要包括内连接、交叉连接、自连接以及外连接查询4种类型。本章主要介绍这4种连接查询的使用方法，在本章的最后还将介绍3种主要的集合查询。

本章重点：

- ☐ 等值连接和非等值连接查询的使用方法
- ☐ 使用ON子句建立相等连接
- ☐ 使用USING子句建立相等连接
- ☐ 交叉连接的使用方法
- ☐ 自连接查询的使用方法
- ☐ 外连接查询的使用方法
- ☐ 集合查询的使用方法

### 8.1 内连接查询

很多时候，需要将几个表连接起来并查询满足条件的记录，这个时候就需要使用内连接。内连接包括等值连接和非等值连接。除了使用WHERE子句中的等号运算符（=）建立等值连接外，还可以使用ON子句和USING子句建立等值连接。这一节将介绍几种内连接的查询方法。

#### 8.1.1 等值连接

等值连接是指将指定的连接条件通过使用等号运算符（=）连接起来，并返回符合连接条件的数据行。其语法格式如下：

```
SELECT 表名1.字段, 表名2.字段 ...  
FROM 表名1,表名2  
WHERE 表名1.字段1=表名2.字段2
```

其中，SELECT 语句中表名1.字段和表名2.字段表示指定数据表1和数据表2中要查询的列；FROM 语句中表名1和表名2表示指定连接的数据表的名字；WHERE子句中表名1.字段1=表名2.字段2表示用于指定连接条件的列。这里的字段1中的列和字段2中的列必须是两个表之间相互关联的列。



**注意** 在进行连接查询时，如果要查询的字段同时存在于两个表当中（也就是说，在不同表之间有相同名字的列），SELECT语句中必须在该列名之前加上表名作为前缀，以便区分到底查询的是哪张表中的字段。

**提示** 如果表之间的列名不相同，在SELECT语句中可以不用在列名的前面加表名作为前缀，但是为了使SQL语句更容易理解，一般情况下，都需要在列名的前面加上表名作为前缀。

### 例8.1 查询学生所选课程的成绩信息。

```
SELECT T_result.stuID,T_curriculum.curID, T_curriculum.curName,T_result.result
FROM T_result,T_curriculum
WHERE T_result.curID=T_curriculum.curID
ORDER BY T_result.stuID ASC
```

这里的连接查询需要用到两张表，分别是成绩信息表（T\_result）和课程信息表（T\_curriculum）。在SELECT语句中分别对成绩信息表中的学生编号和学生课程成绩以及课程信息表中的课程编号和课程的名字进行查询。FROM语句中指定了要进行连接查询的两张表。WHERE子句中指定了连接查询的条件，这里是以成绩信息表和课程信息表中的课程编号作为连接条件进行连接查询的。在SQL语句的最后使用ORDER BY子句按照成绩信息表中的学生编号的升序顺序进行排序操作。其查询结果如图8.1所示。

stuID	curID	curName	result
s102203	i232	数据库基础	75
s102203	i105	计算机系统结构	85
s102203	i333	高等数学	60
s102203	i321	C语言	90
s112303	i333	高等数学	75
s112303	i321	C语言	65
s206363	i105	计算机系统结构	80
s206363	i333	高等数学	55
s221256	i232	数据库基础	60
s2532653	i321	C语言	90
s2532653	i333	高等数学	53
s2532653	i105	计算机系统结构	80

图8.1 查询学生所选课程的成绩信息

在连接查询的过程中，在SELECT语句和WHERE子句中经常会用到表的名字。如果连接的表很多或者表的名字很长，使用表名会很不方便，因此可以在连接查询时使用表别名的方式来简化连接查询的操作，同时使用表别名的方式也可以提高查询的性能。使用表别名进行连接查询的语法规则如下：

```
SELECT A.字段,B.字段 ...
FROM 表名1 A,表名2 B
WHERE A.字段1=B.字段2
```

这里在FROM语句中，表名1后面紧跟了一个字母A，这个字母A就是表名1的别名；表名2后面紧跟了一个字母B，这个字母B就是表名2的别名。在SELECT语句和WHERE子句中就可以直接使用表的别名了。这里还以例8.1中查询学生所选课程的成绩信息的例子为例，看一下如何使用表别名的方式进行连接查询。

### 例8.2 查询学生所选课程的成绩信息（使用表别名）。

```
SELECT R.stuID,C.curID, C.curName,R.result
FROM T_result R,T_curriculum C
WHERE R.curID=C.curID
ORDER BY R.stuID
```

这里使用表别名的方式对学生所选课程的成绩信息进行连接查询。其中，T\_result表的别名定义为R，T\_curriculum表的别名定义为C，在SELECT语句、WHERE子句以及ORDER BY子句中使用R就代表T\_result表，使用C就代表T\_curriculum表。其查询的结果与图8.1中显示的结果是相同的。

**说明** 在连接查询时，使用表别名的方式不仅可以简化连接查询操作，而且还可以提高查询的性能。

在MySQL数据库中，还可以使用INNER JOIN关键字显式的表与表之间进行的是内连接的查询操作。以8.2的例子为例，使用INNER JOIN关键字对两个表进行等值连接的SQL语句如下：

```
SELECT R.stuID, C.curID, C.curName, R.result
FROM T_result R INNER JOIN T_curriculum C
ON R.curID=C.curID
ORDER BY R.stuID
```

这里使用INNER JOIN关键字取代FROM语句中的逗号，使用关键字ON代替WHERE指定要查询的条件。其最后的查询结果与图8.1中显示的结果是相同的。

在等值的连接查询中，还可以在WHERE子句中使用AND关键字指定其他的查询条件，此时查询的结果是既满足连接条件又满足AND关键字指定条件的记录。来看下面这个例子。

**例8.3** 查询学生编号为s102203学生的选课成绩信息。

```
SELECT R.stuID, C.curID, C.curName, R.result
FROM T_result R, T_curriculum C
WHERE R.curID=C.curID
AND R.stuID = 's102203'
```

这条连接查询的语句中使用AND关键字指定了成绩信息表中学生编号为s102203的查询条件，因此最后的查询结果就只会显示学生编号为s102203的选课成绩信息，如图8.2所示。

当然，连接查询并不仅仅局限于两张表之间的连接，三张表或者更多表之间同样可以进行连接操作。例如在前面的两个例子中，查询学生的选课成绩信息只是查询出了学生的编号，并没有查询出学生编号所代表的学生的姓名信息。要想显示学生的具体信息，就还得需要连接一张表，这就是学生信息表（T\_student）。下面来看一个例子。

**例8.4** 查询学生姓名为赵亮的学生选课成绩信息。

```
SELECT R.stuID, S.stuName, C.curID, C.curName, R.result
FROM T_result R, T_curriculum C, T_student S
WHERE R.curID=C.curID
AND R.stuID=S.stuID
AND S.stuName = '赵亮'
```

这里的连接查询需要使用三张表，分别是成绩信息表（T\_result）、课程信息表（T\_curriculum）和学生信息表（T\_student）。在WHERE子句中需要将这三张表都连接起来。其中等值连接R.curID=C.curID是用来将成绩信息表（T\_result）和课程信息表（T\_curriculum）连接起来，等值连接R.stuID=S.stuID是用来将成绩信息表（T\_result）和学生信息表（T\_student）连接起来。通过这两个等值连接的操作，就将需要查询的成绩信息表（T\_result）、课程信息表（T\_curriculum）和学生信息表（T\_student）这三张表连接到了在一起，最后的条件S.stuName = '赵亮'表示要查询的是学生姓名为赵亮的学生选课成绩信息。在WHERE子句中通过AND关键字将需要查询的条件连接起来。通过这样的连接操作，在SELECT语句中就可以查询这三张表中的信息了。其查询结果如图8.3所示。

stuID	curID	curName	result
s102203	i105	计算机系统结构	85
s102203	i232	数据库基础	75
s102203	i321	C语言	90
s102203	i333	高等数学	60

图8.2 查询学生编号为s102203学生的选课成绩信息

stuID	stuName	curID	curName	result
s102203	赵亮	i105	计算机系统结构	85
s102203	赵亮	i232	数据库基础	75
s102203	赵亮	i321	C语言	90
s102203	赵亮	i333	高等数学	60

图8.3 查询学生姓名为赵亮的学生选课成绩信息

**提示** 在等值的连接查询中，如果在WHERE子句中使用AND关键字指定其他的查询条件，那么两个表之间相互关联的列（即使用等号运算符指定连接条件的列）应该放在WHERE子句的前面，以便提供查询效率。

8.1.2 非等值连接

非等值连接是指使用除等号运算符(=)以外的其他运算符将指定条件连接起来而执行的查询操作。其他运算符包括>=（大于等于）、<=（小于等于）、>（大于）、<（小于）、!=（不等于）等，还可以使用BETWEEN...AND运算符。

例8.5 查询课程成绩在80分以上的学生选课信息。

```
SELECT R.stuID,C.curID, C.curName,R.result
FROM T_result R,T_curriculum C
WHERE R. result>80
```

这里的WHERE子句中使用大于运算符(>)在成绩信息表(T\_result)和课程信息表(T\_curriculum)中查询课程成绩在80分以上的学生选课信息。其查询的结果如图8.4所示。

从查询的结果可以看到，使用非等值连接查询出来的结果集中含有大量的重复元素，单独使用非等值连接查询的意义并不大，一般情况下，需要和等值连接等其他查询放到一起使用。例如，要想查询课程成绩在80分以上的学生选课信息，可以使用下面的SQL语句。

stuID	curID	curName	result
s102203	i105	计算机系统结构	85
s102203	i232	数据库基础	85
s102203	i321	C语言	85
s102203	i333	高等数学	85
s102203	i542	操作系统	85
s102203	i105	计算机系统结构	90
s102203	i232	数据库基础	90
s102203	i321	C语言	90
s102203	i333	高等数学	90
s102203	i542	操作系统	90
s2532653	i105	计算机系统结构	90
s2532653	i232	数据库基础	90
s2532653	i321	C语言	90
s2532653	i333	高等数学	90
s2532653	i542	操作系统	90

图8.4 查询课程成绩在80分以上的学生选课信息（非等值连接）

```
SELECT R.stuID,C.curID, C.curName,R.result
FROM T_result R,T_curriculum C
WHERE R.curID=C.curID
AND R. result>80
```

这里的WHERE子句中有两个条件，其中R.curID=C.curID指定了连接查询的条件，这里是以成绩信息表和课程信息表中的课程编号作为连接条件进行连接查询的。在AND关键字后又加上了一个限定条件R. result>80，表示要查询的结果集中学生的课程成绩应该大于80。其运行结果如图8.5所示。

stuID	curID	curName	result
s102203	i105	计算机系统结构	85
s102203	i321	C语言	90
s2532653	i321	C语言	90

图8.5 查询课程成绩在80分以上的学生选课信息

通过执行上面的SQL语句，可以看到此时查询的结果就只要3条记录了。其中编号为s102203的学生有两门课的成绩都达到了80分以上，编号为s2532653的学生有一门课的成绩达到了80分以上。

### 8.1.3 使用ON子句建立相等连接

在SQL语句中除了WHERE子句中使用等号运算符(=)实现等值连接的操作之外，还可以使用ON子句建立相等连接条件。其语法规则如下：

```
SELECT 表名1.字段, 表名2.字段 ...
FROM 表名1 JOIN 表名2
ON 表名1.字段1=表名2.字段
```

其中关键字JOIN表示将表1和表2连接起来，ON子句用来指定连接条件的列。这里的字段1中的列和字段2中的列必须是两个表之间相互关联的列。下面来看一个使用ON子句建立相等连接条件的例子。

**例8.6** 查询学生编号为s102203学生的选课成绩信息（使用ON子句）。

```
SELECT R.stuID,C.curID, C.curName,R.result
FROM T_result R JOIN T_curriculum C
ON R.curID=C.curID
WHERE R.stuID = 's102203'
```

这条连接查询的语句中，使用ON子句建立相等连接条件，WHERE子句中指定成绩信息表中学生编号为s102203的查询条件，其查询结果如图8.6所示。

**说明**

虽然ON子句中也可以使用AND关键字指定其他的查询条件，但是在使用ON子句建立相等连接时，最后在ON关键字后面只跟连接条件，而将其他的限制条件都写到WHERE子句中。

stuID	curID	curName	result
s102203	t105	计算机系统结构	85
s102203	i232	数据库基础	75
s102203	i321	C语言	90
s102203	i333	高等数学	60

图8.6 查询学生编号为s102203学生的选课成绩信息（使用ON子句）

### 8.1.4 使用USING子句建立相等连接

在进行连接操作时，有时只希望将两张表中相互关联的列建立一个等值连接。此时，可以使用USING子句建立相等连接来简化使用等号运算符(=)建立的等值连接操作。其语法规则如下：

```
SELECT 表名1.字段, 表名2.字段 ...
FROM 表名1 JOIN 表名2
USING (字段1)
```

其中关键字JOIN表示将表1和表2连接起来，USING子句中使用括号将字段1括起来，字段1就是两个表中建立等值连接相互关联的列。下面来看一个使用USING子句建立相等连接的例子。

**例8.7** 查询学生所选课程的成绩信息（使用USING子句）。

```
SELECT R.stuID,C.curID, C.curName,R.result
FROM T_result R JOIN T_curriculum C
USING (curID)
```

这条连接查询的语句中，使用USING子句建立相等连接，这里的curID字段就是成绩信息表（T\_result）和课程信息表（T\_curriculum）相互关联的列。其查询结果如图8.7所示。

stuID	curID	curName	result
s102203	t105	计算机系统结构	85
s206363	t105	计算机系统结构	90
s2532653	t105	计算机系统结构	50
s102203	i232	数据库基础	75
s221256	i232	数据库基础	60
s102203	i321	C语言	90
s112303	i321	C语言	65
s2532653	i321	C语言	90
s102203	i333	高等数学	60
s112303	i333	高等数学	75
s206363	i333	高等数学	55
s2532653	i333	高等数学	53

图8.7 查询学生所选课程的成绩信息（使用USING子句）

从这个查询结果可以看到，使用USING子句建立相等连接和使用等号运算符(=)建立的同连接查询出来的结果是一样的。只是查询出来的记录没有经过排序，如果想对查询的结果进行排序操作，可以在该SQL的语句最后加上一个ORDER BY子句。例如，如果想对查询的结果按照stuID（学生编号）排序，这个SQL语句就可以这样完成。

```
SELECT R.stuID,C.curID, C.curName,R.result
FROM T_result R JOIN T_curriculum C
USING (curID)
ORDER BY R.stuID
```

8.2 交叉连接

交叉连接返回的结果是一个笛卡儿积。所谓笛卡儿积，实际就是两个集合相乘的结果。假设集合A中有n个元素，集合B中有m个元素，如果最后返回的结果是n\*m，那么这个结果就是集合A和集合B的笛卡儿积。下面通过图的形式来直观地介绍一下什么是笛卡儿积。

图8.8中有两张表，分别是学生信息表（T\_student）和成绩信息表（T\_result），在学生信息表中有4条记录，在成绩信息表中有2条记录。

stuID	stuName
s102203	赵亮
s112303	郑磊
s115263	王海
s206363	张明

stuID	curID	result
s102203	t105	85
s206363	t333	55

图8.8 学生信息表（T\_student）和成绩信息表（T\_result）

这两张表如果进行交叉连接，根据上面讲到的内容，应该返回的就是一个笛卡儿积。也就是应该返回的是4×2=8条记录。其结果如图8.9所示。

从这个查询结果中可以看到，学生信息表中学生编号为s102203的学生，既和成绩信息表中课程编号为t105的课程成绩信息组成一条记录，又和成绩信息表中课程编号为t333的课程成绩信息组成另一条记录。同样道理，学生信息表中学生编号为s112303的学生也要和课程编号为t105和课程编号为t333的课程成绩信息组成两条记录，以此类推，按照这样的连接方式，最后显示的就是有8条记录的这样一个结果集。

stuID	stuName	age	stuID	curID	result
s102203	赵亮	23	s102203	t105	85
s102203	赵亮	23	s206363	t333	55
s112303	郑磊	21	s102203	t105	85
s112303	郑磊	21	s206363	t333	55
s115263	王海	23	s102203	t105	85
s115263	王海	23	s206363	t333	55
s206363	张明	22	s102203	t105	85
s206363	张明	22	s206363	t333	55

图8.9 T\_student表和T\_result表交叉连接后的结果

如果表与表之间进行连接查询时没有在WHERE子句中指定查询条件，那么这两个表做的就是一个交叉连接，经过交叉连接返回的结果集的行数实际是两个表行数的乘积。来看下面这个例子。

例8.8 对成绩信息表（T\_result）和课程信息表（T\_curriculum）进行交叉连接。

```
SELECT R.stuID,C.curID
FROM T_result R,T_curriculum C
```

这段SQL语句中没有WHERE子句指定查询条件，是对成绩信息表（T\_result）和课程信息表（T\_curriculum）进行交叉连接查询。以下是执行完交叉连接后的查询结果。

+-----+-----+
---------------



```
| stuID | curID |
+-----+-----+
| s102203 | t105 |
| s102203 | t232 |
| s102203 | t321 |
| s102203 | t333 |
| s102203 | t542 |
| s102203 | t105 |
| s102203 | t232 |
| s102203 | t321 |
| s102203 | t333 |
| s102203 | t542 |
| ... | ... |
+-----+-----+
65 rows in set
```

这条SQL语句一共产生了65条记录。根据对笛卡儿积的介绍，读者可以考虑一下，对成绩信息表（T\_result）和课程信息表（T\_curriculum）这两张表进行交叉连接，其返回的结果集的行数为什么会是65。

**提示** 在成绩信息表（T\_result）中一共有13条记录，在课程信息表（T\_curriculum）中一共有5条记录。

另外，SQL语句中还提供了另外一种建立交叉连接的方法，这就是使用CROSS JOIN关键字。通过使用CROSS JOIN关键字建立的两张表的连接最后产生的结果也是这两张表的笛卡儿积。例如例8.8的这个例子，使用CROSS JOIN关键字建立交叉连接的SQL语句如下：

```
SELECT R.stuID, C.curID, C.curName, R.result
FROM T_result R CROSS JOIN T_curriculum C
```

这里使用CROSS JOIN关键字取代FROM语句中的逗号。使用CROSS JOIN关键字显式表示对两张表进行交叉连接的操作，返回的结果集的行数是两个表行数的乘积。该语句最后查询的结果和例8.8中查询的结果相同。

**注意**

一般情况下，不会使用交叉连接进行数据查询，因为交叉连接中会有大量的重复数据。例如对于两个含有1000行记录的数据表来说，如果使用交叉连接，则会产生1000×1000行的数据。这样不仅会严重影响程序的允许速度和性能，而且其中产生的很多数据并不是开发人员或者是用户希望得到的。因此一般在两张表建立连接查询时，都会使用WHERE子句对查询的条件进行限制，将不满足条件的数据过滤掉。

## 8.3 自连接查询

前面讲到的连接都是在表与表之间进行的。连接查询除了可以在不同的表之间进行，也可以对同一张表实现连接操作，这种连接查询的方式称为自连接。所谓自连接，就是指一个数据表与其自身进行连接。其语法规则如下：

```
SELECT A.字段, A.字段 ...
```



```
FROM 表名1 A,表名1 B
WHERE A.字段=B.字段
```

由于连接的是同一张表，所以在FROM语句中需要为表定义不同的别名，这里为表1分别定义了表的别名为A和B。SELECT语句中可以使用A.字段的形式也可以使用B.字段的形式查询需要的记录。这里的SELECT语句中使用的是A.字段的形式。下面来看一个使用自连接的例子。

**例8.9** 在课程信息表中选择学分数比操作系统的学分数多的课程信息。

```
SELECT C2.curID,C2.curName,C2.credit
FROM T_curriculum C1,T_curriculum C2
WHERE C1.curName = '操作系统'
AND C1.credit<C2.credit
```

这里的SQL语句中是对课程信息表（T\_curriculum）做了自连接的操作。下面来分析一下这段SQL语句产生的过程。

(1) 这段SQL语句首先为课程信息表C1和课程信息表C2做一个笛卡儿乘积。课程信息表中共有5条记录，因此这个笛卡儿乘积的结果应该有25条记录。在这25条记录中，与操作系统这门课有关的记录共有5条（这里只选取课程信息表中的课程编号、课程名和学分数三个字段）。其中，左面的前3个字段表示的是课程信息表C1中的字段信息，后3个字段表示的是课程信息表C2中的字段信息。

curID	curName	credit	curID	curName	credit
t542	操作系统	3	t105	计算机系统结构	4
t542	操作系统	3	t232	数据库基础	5
t542	操作系统	3	t321	C语言	6
t542	操作系统	3	t333	高等数学	4
t542	操作系统	3	t542	操作系统	3

(2) 为了要查询课程信息表中学分数比操作系统的学分数多的课程信息，在上面讲述的查询的基础上，还需要对学分数进行进一步的限制。也就是说需要对表示学分数credit这个字段进行比较。从上面产生的结果可以看到，要想查询出学分数比操作系统的学分数多的课程信息，这里只要让课程信息表C1中的表示学分的字段credit小于课程信息表C2中的表示学分的字段credit就可以了。因此这里在WHERE子句中再加一个AND关键字，增加C1.credit<C2.credit这个限制条件。

curID	cuName	cred
t105	计算机系统结构	4
t232	数据库基础	5
t321	C语言	6
t333	高等数学	4

图8.10 在课程信息表中选择课时数比操作系统的课时数多的课程信息

通过以上两步，就可以查询出在课程信息表中学分数比操作系统的学分数多的课程信息了。其查询的结果如图8.10所示。

**注意** 在自连接时，由于是对同一张数据表中的内容进行连接查询，所以在FROM语句中需要为表定义不同的别名。

8.4 外连接查询

在前面讲述的连接操作中，返回的结果都是满足连接条件的记录。有些时候，开发人员或者用户

对于不满足连接条件的部分记录也感兴趣，这个时候就需要使用外连接查询。外连接查询不仅可以返回满足连接条件的记录，对于一个数据表中在另一个数据表中不匹配的记录也可以返回。外连接查询主要包括3种：左外连接、右外连接和全外连接。这一节将介绍这3种连接在不同数据库中的使用方法。

### 8.4.1 左外连接

左外连接查询的结果中不仅将显示满足连接条件的记录，而且还包括左侧表中不满足查询条件的记录。下面分别介绍在Oracle数据库、MySQL数据库和Microsoft SQL Server数据库中左外连接的使用方法。

#### 1. Oracle数据库

在Oracle数据库中，可以使用加号运算符（+）来表示左外连接。当该加号运算符（+）出现在连接条件的左边时，就称之为左外连接。其语法格式如下：

```
SELECT 表名1.字段, 表名2.字段 ...  
FROM 表名1,表名2  
WHERE 表名1.字段1(+)=表名2.字段2
```

其中，SELECT 语句中表名1.字段和表名2.字段表示指定数据表1和数据表2中要查询的列；FROM 语句中表名1和表名2表示指定连接的数据表的名字；WHERE子句中表名1.字段1(+)=表名2.字段2表示左外连接。此时，表名1.字段1所在的列的值将会被全部查询出来。

例8.10 查询学生对应的所有选课信息。

```
SELECT R.stuID,R.curID,C.curName  
FROM T_result R ,T_curriculum C  
WHERE R.curID(+)=C.curID
```

这段SQL语句查询学生对应的所有选课信息。其中，在WHERE子句中使用了左外连接，其查询结果如下所示。

stuID	CURid	curName
s102203	t105	计算机系统结构
s102203	t232	数据库基础
s102203	t321	C语言
s102203	t333	高等数学
s112203	t321	C语言
s112303	t333	高等数学
s206363	t105	计算机系统结构
s206363	t333	高等数学
s221256	t232	数据库基础
s2532653	t232	
s2532653	t105	计算机系统结构
s2532653	t321	C语言
s2532653	t333	高等数学

这条SQL语句中将左侧的成绩信息表（T\_result）中的学生记录全部查询出来，也包括不满足课程号连接条件的记录。

**注意** 在执行左外连接操作时，如果WHERE子句中有多个限制条件，则每一个限制条件中都应该使用加号运算符 (+)。外连接操作不能用于OR和IN运算符，也不能在子查询中使用。

**说明** 使用加号运算符 (+) 来实现左外连接，适合在Oracle数据库中使用，但在MySQL和Microsoft SQL Server数据库中并不适用。

2. MySQL和Microsoft SQL Server数据库

在MySQL数据库和Microsoft SQL Server数据库中可以使用LEFT[OUTER] JOIN关键字实现，其中OUTER关键字是可选的。使用LEFT[OUTER] JOIN关键字实现左外连接的语法规则如下：

```
SELECT 表名1.字段, 表名2.字段 ...
FROM 表名1 LEFT JOIN表名2
ON 表名1.字段1=表名2.字段2
```

这里使用LEFT JOIN关键字代替SQL语句中FROM语句里的逗号，使用ON子句代替标准SQL语句中的WHERE子句，并将SQL语句中表示左外连接的加号运算符 (+) 去除。下面以例8.10为例，看一下在MySQL数据库中如何实现左外连接。

```
SELECT R.stuID,R.curID,C.curName
FROM T_result R LEFT JOIN T_curriculum C
on R.curID=C.curID
```

其查询的结果如图8.11所示。

在执行左外连接的操作中，还可以使用8.1.4小节讲到的USING子句来简化其操作。以上面的例子为例，使用USING子句实行左外连接的SQL语句如下：

```
SELECT R.stuID,R.curID,C.curName
FROM T_result R LEFT JOIN T_curriculum C
USING(curID)
```

stuID	curID	curName
s102203	t105	计算机系统结构
s102203	t232	数据库基础
s102203	t321	C语言
s102203	t333	高等数学
s112303	t321	C语言
s112303	t333	高等数学
s206363	t105	计算机系统结构
s206363	t333	高等数学
s221256	t232	数据库基础
s2532653	t232	数据库基础
s2532653	t105	计算机系统结构
s2532653	t321	C语言
s2532653	t333	高等数学

图8.11 查询学生对应的所有选课信息

8.4.2 右外连接

右外连接中查询的结果中不仅将显示满足连接条件的记录，而且还包括右侧表中不满足查询条件的记录。下面分别介绍在Oracle数据库、MySQL数据库和Microsoft SQL Server数据库中右外连接的使用方法。

1. Oracle数据库

在Oracle数据库中，当该加号运算符 (+) 出现在连接条件的右边时，就称之为右外连接。其语法格式如下：

```
SELECT 表名1.字段, 表名2.字段 ...
FROM 表名1,表名2
WHERE 表名1.字段1=表名2.字段2(+)
```

其中，SELECT 语句中表名1.字段和表名2.字段表示指定数据表1和数据表2中要查询的列；FROM 语句中表名1和表名2表示指定连接的数据表的名字；WHERE子句中表名1.字段1=表名2.字段2(+)表示



右外连接。此时，表名2.字段2所在的列的值将会被全部查询出来。

**例8.11** 查询所有课程对应的学生信息。

```
SELECT R.stuID,R.curID,C.curName
FROM T_result R ,T_curriculum C
WHERE R.curID=C.curID(+)
```

这段SQL语句查询所有课程对应的学生信息。其中，在WHERE子句中使用了右外连接，其查询结果如下所示。

stuID	CURid	curName
s102203	t105	计算机系统结构
s206363	t105	计算机系统结构
s2532653	t105	计算机系统结构
s102203	t232	数据库基础
s221256	t232	数据库基础
s102203	t321	C语言
s102203	t321	C语言
s2532653	t321	C语言
s102203	t333	高等数学
s112303	t333	高等数学
s206363	t333	高等数学
s2532653	t333	高等数学
		操作系统

这条SQL语句中将右侧的课程信息表（T\_curriculum）中的所有课程记录全部查询出来，也包括学生没有选择的课程记录，这里的操作系统课程就是所有学生都没有选择的课程。

**注意** 在执行右外连接操作时，如果WHERE子句中有多个限制条件，则每一个限制条件中都应该使用加号运算符（+）。

使用加号运算符（+）来实现右外连接，适合在Oracle数据库中使用，但在MySQL和Microsoft SQL Server数据库中并不适用。

## 2. MySQL和Microsoft SQL Server数据库

在MySQL和Microsoft SQL Server数据库中使用RIGHT [OUTER] JOIN关键字实现，其中OUTER关键字是可选的。使用RIGHT [OUTER] JOIN关键字实现右外连接的语法规则如下：

```
SELECT 表名1.字段, 表名2.字段 ...
FROM 表名1 RIGHT JOIN表名2
ON 表名1.字段1=表名2.字段2
```

这里使用RIGHT JOIN关键字代替SQL语句中FROM语句里的逗号，使用ON子句代替SQL语句中的WHERE子句，并将标准SQL语句中表示右外连接的加号运算符（+）去除。下面以例8.11为例，看一下在MySQL数据库中如何实现右外连接。

```
SELECT R.stuID,R.curID,C.curName
FROM T_result R RIGHT JOIN T_curriculum C
on R.curID=C.curID
```



其查询的结果如图8.12所示。

在执行右外连接的操作中，还可以使用8.1.4节讲到的USING子句来简化其操作。以上面的例子为例，使用USING子句实行右外连接的SQL语句如下：

```
SELECT R.stuID,R.curID,C.curName
FROM T_result R RIGHT JOIN T_curriculum C
USING(curID)
```

stuID	curID	curName
s102203	t105	计算机系统结构
s206363	t105	计算机系统结构
s2532653	t105	计算机系统结构
s102203	t232	数据库基础
s221256	t232	数据库基础
s102203	t321	C语言
s112303	t321	C语言
s2532653	t321	C语言
s102203	t333	高等数学
s112303	t333	高等数学
s206363	t333	高等数学
s2532653	t333	高等数学
操作系统		操作系统

图8.12 查询所有课程对应的学生信息

### 8.4.3 全外连接

全外连接中查询的结果中不仅将显示左侧表中满足连接条件的记录，而且还会显示右侧表中不满足查询条件的记录。全外连接可以认为是左外连接与右外连接的合集（不包括重复行）。

全外连接可以使用FULL [OUTER] JOIN关键字实现，其中OUTER关键字是可选的。使用FULL [OUTER] JOIN关键字实现全外连接的语法规则如下：

```
SELECT 表名1.字段, 表名2.字段 ...
FROM 表名1 FULL JOIN 表名2
ON 表名1.字段1=表名2.字段2
```

这里使用FULL JOIN关键字代替SQL语句中FROM语句里的逗号，使用ON子句代替SQL语句中的WHERE子句。下面看一个使用FULL [OUTER] JOIN关键字实现全外连接的例子。

例8.12 查询所有学生的所有选课信息。

```
SELECT R.stuID,R.curID,C.curName
FROM T_result R FULL JOIN T_curriculum C
on R.curID=C.curID
```

stuID	CURid	curName
s102203	t105	计算机系统结构
s102203	t232	数据库基础
s102203	t321	C语言
s102203	t333	高等数学
s112203	t321	C语言
s112303	t333	高等数学
s206363	t105	计算机系统结构
s206363	t333	高等数学
s221256	t232	数据库基础
s2532653	t232	
s2532653	t105	计算机系统结构
s2532653	t321	C语言
s2532653	t333	高等数学
		操作系统

这条SQL语句中将左侧的成绩信息表（T\_result）和右侧的课程信息表（T\_curriculum）中的所有课程记录全部查询出来。这个结果集是一个左外连接和右外连接的合集，而且不包括重复行。





**注意** 这里给出的SQL语句适合Oracle数据库，但并不适合于MySQL数据库。MySQL数据库是不支持全外连接的。可以通过执行并操作（UNION）的方法对左外连接和右外连接求合集，取得和全外连接相同的查询结果。有关并操作（UNION）的查询方法请参考8.5.1节。

## 8.5 集合查询

在SQL的连接查询语句中，还有一种查询方式就是集合查询。集合查询主要包括3种：并操作、交操作和差操作。其中交操作和差操作并不是对目前主流的所有的数据库都适用。这一节中将分别介绍这3种集合的操作方法。

### 8.5.1 并操作

执行并操作（UNION）使用的关键字是UNION。并操作返回的结果集是包括了两个查询语句中查询出来的所有不同的行，不包含重复行。其语法格式如下：

```
SELECT 语句1
UNION
SELECT 语句2
```

其中语句1和语句2表示的是两个用于查询的SELECT语句。UNION关键字表示对这两个查询语句查询出来的结果进行并操作。这里需要保证SELECT 语句1和SELECT 语句2中查询出的列数必须相同，而且对应的列的数据类型必须一致。下面看一个使用UNION关键字实现并操作的例子。

**例8.13** 查询所有学生的所有选课信息（使用并操作）。

```
SELECT R.stuID,R.curID,C.curName
FROM T_result R LEFT JOIN T_curriculum C
on R.curID=C.curID
UNION
SELECT R.stuID,R.curID,C.curName
FROM T_result R RIGHT JOIN T_curriculum C
on R.curID=C.curID
```

这段SQL语句中使用UNION关键字连接两个SELECT语句，并将这两个SELECT语句查询到的结果合并到一起作为结果集返回。其查询的结果如图8.13所示。

在MySQL 5.0数据库中执行并操作，可以实现类似全外连接的查询结果。使用UNION关键字实现并操作会自动去除重复的行，如果希望保留重复的记录，可以使用UNION ALL关键字。

**注意** 在进行并操作时，两个SELECT语句中要查询列对应的属性的个数和数据类型必须是相同的。如果两个SELECT语句中查询的列数不相同或者对应列的数据类型不相同，则不能进行并操作，此时执行的SQL语句会报错。

stuID	curID	curName
s102203	i105	计算机系统结构
s102203	i232	数据库基础
s102203	i321	C语言
s102203	i333	高等数学
s112303	i321	C语言
s112303	i333	高等数学
s206363	i105	计算机系统结构
s206363	i333	高等数学
s221256	i232	数据库基础
s2532653	i232	数据库
s2532653	i105	计算机系统结构
s2532653	i321	C语言
s2532653	i333	高等数学
data	data	操作系统

图8.13 查询所有学生的所有选课信息（使用并操作）

### 8.5.2 交操作

执行交操作（INTERSECT）使用的关键字是INTERSECT。交操作返回的结果集包括了连接查询结果的公共行。交操作中不会出现重复行。其语法格式如下：

```
SELECT 语句1  
INTERSECT  
SELECT 语句2
```

其中语句1和语句2表示的是两个用于查询的SELECT语句。INTERSECT关键字表示对这两个查询语句查询出来的结果进行交操作。这里需要保证SELECT 语句1和SELECT 语句2中查询出的列数必须相同，而且对应的列的数据类型必须一致。

例如，有两个表A和表B，表A中有3个列，分别是A1、A2和A3，表B中也包含三个列，分别是B1、B2和B3。这里假设字段A1和字段B1有相同的数据类型，字段A2和字段B2有相同的数据类型，字段A3和字段B3有相同的数据类型。现在对这两张表执行交操作。这两张表的结构如图8.14所示。

A1	A2	A3
a	b	c
d	e	f
g	h	i
j	k	l

B1	B2	B3
m	n	o
p	q	r
a	b	c
j	k	l

图8.14 执行交操作的表A和表B

对这两张表中的记录进行交操作。这里表A查询的列为列A1、A2和A3，表B查询的列为列B1、B2和B3。使用INTERSECT关键字对查询结果进行交操作。根据交操作的运算规则，执行A交B的运算之后，其执行的结果如图8.15所示。

a	b	c
j	k	l

图8.15 表A与表B执行交操作的结果

了解了交操作的运算规则之后，下面通过一个例子来看一下如何在数据库的查询中运用INTERSECT关键字。

**例8.14** 对教师信息表（T\_teacher）和计算机系教师信息表（T\_CSteacher）实行交操作。

```
(SELECT teaID, teaName, dept, profession  
FROM T_teacher)  
INTERSECT  
(SELECT teaID, teaName, dept, profession  
FROM T_CSteacher)
```

这段SQL语句要对教师信息表（T\_teacher）和计算机系教师信息表（T\_CSteacher）实行交操作。其中，教师信息表（T\_teacher）包含有全校教师的信息，计算机系教师信息表（T\_CSteacher）中包含有所有计算机系的教师信息。其查询结果如下所示。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t103265	张昌	计算机系	教授
t105320	于波	计算机系	讲师
t106358	毛翠	计算机系	教授

这里的SQL语句中查询出来的结果是在教师信息表（T\_teacher）和计算机系教师信息表（T\_CSteacher）中都存在的教师信息，即所有计算机系的教师信息。下面来分析一下这段SQL语句产生的过程。



(1) 这段SQL语句包括了两个SELECT查询语句，第一个SELECT查询语句是用来查询教师信息表 (T\_teacher) 的教师信息。其SELECT查询语句查询的结果如下所示。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t103265	张昌	计算机系	教授
t105320	于波	计算机系	讲师
t106358	毛翠	计算机系	教授
t156354	王新	数学系	讲师
t156355	李中	数学系	教授
t181585	李慧	物理系	教授
t186585	孙立	物理系	讲师

这段SELECT查询语句中查询结果包括了教师信息表中所有的教师信息。

(2) 第二个SELECT查询语句是用来查询计算机系教师信息表 (T\_CSteacher) 的教师信息。其SELECT查询语句查询的结果如下所示。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t103265	张昌	计算机系	教授
t105320	于波	计算机系	讲师
t106358	毛翠	计算机系	教授

这段SELECT查询语句中查询结果包括了计算机系教师信息表中所有计算机系的教师信息。

(3) 使用INTERSECT关键字对这两段SQL语句查询出来的结果执行交操作。其返回的查询结果集应该是上述两段SQL语句查询结果公共行。从上述两段SQL语句查询结果中可以看出，结果的公共行就应该是计算机系教师的信息。

使用INTERSECT关键字实现交操作会自动去除重复的行，如果希望保留重复的记录，可以使用INTERSECT ALL关键字。

**说明** INTERSECT关键字可以在Oracle数据库和Microsoft SQL Server数据库中使用。MySQL数据库中不支持INTERSECT关键字。

**注意** 在进行交操作时，两个SELECT语句中要查询列对应的属性的个数和数据类型必须是相同的。如果两个SELECT语句中查询的列数不相同或者对应列的数据类型不相同，则不能进行交操作，此时执行的SQL语句会报错。

8.5.3 差操作

执行交操作 (MINUS) 使用的关键字是MINUS。差操作返回的记录结果集只在第一个SELECT语句中出现，但不存在于第二个SELECT语句的查询结果中。在执行差操作时，首先会找出第一个SELECT语句中产生的结果集，然后再将这些结果集和第二个SELECT语句中查询的结果集进行比较，如果结果集中的记录存在于第二个SELECT语句中查询的结果集，则将这些记录舍弃，最后的结果集中只会出现在第一个SELECT语句查询出的结果集中有而不在第二个SELECT语句查询的结果集的记录。

其语法格式如下：

```
SELECT 语句1
MINUS
SELECT 语句2
```

其中语句1和语句2表示的是两个用于查询的SELECT语句。MINUS关键字表示对这两个查询语句查询出来的结果进行差操作。这里需要保证SELECT 语句1和SELECT 语句2中查询出的列数必须相同，而且对应的列的数据类型必须一致。

例如，有两个表A和表B，表A中有3个列，分别是A1、A2和A3，表B中也包含3个列，分别是B1、B2和B3。这里假设字段A1和字段B1有相同的数据类型，字段A2和字段B2有相同的数据类型，字段A3和字段B3有相同的数据类型。现在对这两张表执行差操作。这两张表的结构如图8.14所示。

d	e	f
g	h	k

图8.16 表A与表B执行差操作的结果

对这两张表中的记录进行差操作。这里表A查询的列为列A1、A2和A3，表B查询的列为列B1、B2和B3。使用MINUS关键字对查询结果进行差操作。根据差操作的运算规则，执行A差B的运算之后，其执行的结果如图8.16所示。

了解了差操作的运算规则之后，下面通过一个例子来看一下如何在数据库的查询中运用MINUS关键字。

**例8.15** 对教师信息表（T\_teacher）和计算机系教师信息表（T\_CSteacher）实行差操作。

```
(SELECT teaID, teaName, dept, profession
FROM T_teacher)
MINUS
(SELECT teaID, teaName, dept, profession
FROM T_CSteacher)
```

这段SQL语句要对教师信息表（T\_teacher）和计算机系教师信息表（T\_CSteacher）实行差操作。其中，教师信息表（T\_teacher）包含有全校教师的信息，计算机系教师信息表（T\_CSteacher）中包含有所有计算机系的教师信息。其查询结果如下所示。

teaID	teaName	dept	profession
t156354	王新	数学系	讲师
t156355	李中	数学系	教授
t181585	李慧	物理系	教授
t186585	孙立	物理系	讲师

这里的SQL语句中查询出来的结果是在教师信息表（T\_teacher）中存在但是在计算机系教师信息表（T\_CSteacher）中不存在的教师信息，即显示的是除了计算机系以外其他院系的教师信息。下面来分析一下这段SQL语句产生的过程。

(1) 这段SQL语句包括了两个SELECT查询语句，第一个SELECT查询语句是用来查询教师信息表（T\_teacher）的教师信息。其SELECT查询语句查询的结果如下所示。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t103265	张昌	计算机系	教授
t105320	于波	计算机系	讲师



## 零基础学SQL

t106358	毛翠	计算机系	教授
t156354	王新	数学系	讲师
t156355	李中	数学系	教授
t181585	李慧	物理系	教授
t186585	孙立	物理系	讲师

这段SELECT查询语句中查询结果包括了教师信息表中所有的教师信息。

(2) 第二个SELECT查询语句是用来查询计算机系教师信息表（T\_CSteacher）的教师信息。其SELECT查询语句查询的结果如下所示。

teaID	teaName	dept	profession
t102225	赵伟	计算机系	副教授
t103265	张昌	计算机系	教授
t105320	于波	计算机系	讲师
t106358	毛翠	计算机系	教授

这段SELECT查询语句中查询结果包括了计算机系教师信息表中所有计算机系的教师信息。

(3) 使用MINUS关键字对这两段SQL语句查询出来的结果执行差操作。其返回的查询记录应该是在第一个SELECT语句查询出的结果集中有而在第二个SELECT语句查询的结果集中没有的记录。从上述两段SQL语句查询结果中可以看出，其最终显示的结果集是在第一个SELECT查询语句中存在而在第二个SELECT查询语句中不存在的教师信息。

在Microsoft SQL Server数据库中可以使用EXCEPT关键字完成差操作。使用INTERSECT关键字实现差操作会自动去除重复的行。如果希望保留重复的记录，可以使用INTERSECT ALL或者EXCEPT ALL关键字。

**说明** 在Oracle数据库中可以使用MINUS关键字完成差操作，在Microsoft SQL Server数据库中可以使用EXCEPT关键字完成差操作。MySQL数据库中不支持MINUS关键字和EXCEPT关键字。

**注意** 在进行差操作时，两个SELECT语句中要查询列对应的属性的个数和数据类型必须是相同的。如果两个SELECT语句中查询的列数不相同或者对应列的数据类型不相同，则不能进行差操作，此时执行的SQL语句会报错。

## 8.6 小结

这一章主要介绍了连接查询的使用方法。连接查询中经常用到的是内连接和外连接的操作。除了掌握使用WHERE子句建立连接条件的方法之外，还应该会使用JOIN、ON、USING、LEFT UNION、RIGHT UNION、FULL UNION等关键字实现连接查询。

另外，还应该了解三种主要的集合查询：并操作（UNION）、交操作（INTERSECT）以及差操作（MINUS）的含义。在集合操作中，要特别注意的是两个SELECT语句中要查询列对应的属性的个数和数据类型必须是相同的。否则执行的SQL语句就会出现错误。这三种集合操作并不完全不适用于目前流行的主要的数据库，有些数据库中是不允许使用交操作（INTERSECT）和差操作（MINUS）的，在应用集合操作时，需要了解自己使用的数据库是否支持该集合操作。



## 第9章 子 查 询

所谓子查询，是指将一个SELECT查询语句块嵌套在另一个SQL查询语句中。由于子查询是嵌套在其他的SQL查询语句中，所以也称之为嵌套查询。在SQL语句中，子查询需要放在圆括号中，在执行子查询时，其执行的过程是先查询出来子查询的结果，然后将子查询返回的结果作为其外层查询的查询条件。

子查询根据返回的结果，可以分为单行子查询，多行子查询和多列子查询；根据返回的数据与外层查询之间的关系，可以分为相关子查询和不相关子查询。子查询除了应用在查询语句中，还可以应用在表格创建中用来实现数据表之间的复制。应该说，子查询在实际开发过程中有着非常广泛的应用。本章将全面介绍各种子查询的作用以及它们的使用方法。

本章重点：

- ☐ 在子查询中使用比较运算符
- ☐ 在子查询中使用IN运算符
- ☐ 在子查询中使用ANY运算符
- ☐ 在子查询中使用ALL运算符
- ☐ 多列子查询
- ☐ 相关子查询
- ☐ 子查询在SQL语句中的应用
- ☐ 多重子查询
- ☐ 子查询在CREATE TABLE语句中的应用

### 9.1 单行子查询

在实际应用中，如果开发人员或者是用户明确知道其SQL语句中使用的子查询返回的结果是一行数据，即子查询中返回的结果是一个值时，就可以使用算术比较运算符进行子查询的操作。其中，比较运算符包括=（等于）、>=（大于等于）、<=（小于等于）、>（大于）、<（小于）、!=（不等于）、<>（不等于）、!>（不大于）、!<（不小于）。下面来看一个使用比较运算符进行子查询的例子。

**例9.1** 查询教师信息表中比赵伟老师工资高的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession,salary
FROM T_teacher
WHERE salary >
(SELECT salary
FROM T_teacher
WHERE teaName = '赵伟')
ORDER BY salary ASC
```

## 零基础学SQL

这里要查询的是比赵伟老师工资高的教师信息，使用子查询首先将赵伟老师的工资查询出来，可以很明确地知道该子查询中返回的结果是一个值。在外层查询的WHERE子句中使用大于(>)运算符查询教师工资中比赵伟老师工资高的教师信息。最后使用ORDER BY子句对查询的结果进行升序排序。其查询的结果如图9.1所示。

teaID	teaNa	age	sex	dept	profession	salary
t186585	孙立	48	男	物理系	讲师	3200
t181585	李慧	40	女	物理系	教授	3500
t103265	张昌	43	男	计算机系	教授	3800
t106358	毛翠	50	女	计算机系	教授	4000
t156355	李中	55	女	数学系	教授	4200

图9.1 查询教师信息表中比赵伟老师工资高的教师信息

这里一共查询出了5条记录。在数据库中赵伟老师的工资是3000元。可以看到从数据库中查询出来的结果中，教师的工资都是高于3000元的。下面来分析一下这个结果产生的过程。

(1) 这段SQL语句首先执行的是子查询中的语句，在例9.1中子查询的SQL语句如下：

```
SELECT salary
FROM T_teacher
WHERE teaName = '赵伟'
```

这条子查询语句查询的是教师姓名为赵伟的工资。其查询结果如下所示。

```
salary
-----
3000
```

这里的子查询中查询出教师姓名为赵伟的工资为3000元。

(2) 将子查询的结果3000元作为外层查询的查询条件。在外层查询的WHERE子句中通过使用大于运算符(>)查询出工资大于3000元的教师信息，并将查询到的结果集返回。

(3) ORDER BY子句放在查询语句的最后，对产生的结果按照工资由低到高升序的顺序排序。其最终查询出来的结果集就是图9.1中显示的结果。

**注意** 在使用比较运算符的子查询时，子查询语句要放在比较运算符的后面。如果子查询中返回的数据是多行的，则不能使用比较运算符。

## 9.2 多行子查询

所谓多行子查询，是指子查询中返回的结果集中含有多行数据。当子查询返回的是多行数据时，需要使用多行运算符。多行运算符包括IN、ANY、ALL等运算符。这一节就来介绍多行运算符IN、ANY、ALL在子查询中的使用方法。

### 9.2.1 使用IN运算符的子查询

IN运算符在6.1.3小节中已经讲过。使用IN运算符，可以将满足列表中满足指定表达式的任何一个值都查询出来。在子查询中使用IN运算符，则与子查询中查询出来的结果集中的任何一个值匹配的结果都会被查询出来。因此IN运算符可以用于多行子查询中。下面来看一个使用IN运算符进行子查询的例子。

**例9.2** 查询和教师姓名为毛翠的老师在同一个系的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession,salary
```

```
FROM T_teacher
WHERE dept IN
(SELECT dept
FROM T_teacher
WHERE teaName = '毛翠')
ORDER BY salary ASC
```

这里要查询和教师姓名为毛翠的老师在同一个系的教师信息。首先需要在子查询中查询出教师毛翠所在的院系。然后在外层查询WHERE子句中使用IN运算符对子查询中查询出来的任意一个值进行匹配，并将匹配的结果作为最终查询的结果集显示出来。最后使用ORDER BY子句对查询的结果进行升序排序。其查询结果如图9.2所示。

teaID	teaName	age	sex	dept	profession	salary
t106320	于波	28	男	计算机系	讲师	2800
t102225	赵伟	38	男	计算机系	副教授	3000
t103265	张晶	43	男	计算机系	教授	3800
t106358	毛翠	50	女	计算机系	教授	4000

图9.2 查询和教师姓名为毛翠的老师在同一个系的教师信息

这里查询的结果中一共有4条记录，这4条记录的教师所在的院系都是计算机系的，和教师姓名为毛翠的老师在同一个院系。下面来分析一下这个结果产生的过程。

(1) 这段SQL语句首先执行的是子查询中的语句，在例9.2中子查询的SQL语句如下：

```
SELECT dept
FROM T_teacher
WHERE teaName = '毛翠'
```

这条子查询语句查询的是教师姓名为毛翠的教师所在的院系。其查询结果如下所示。

```
dept
-----
计算机系
```

这里的子查询中查询出教师姓名为毛翠的教师所在的院系是计算机系。

(2) 将子查询的结果计算机系作为外层查询的查询条件。在外层查询的WHERE子句中通过使用IN运算符查询出所在院系是计算机系的教师信息，并将查询到的结果集返回。其最终查询出来的结果集就是图9.2中显示的结果。

(3) ORDER BY子句放在查询语句的最后，对产生的结果按照工资由低到高升序的顺序排序。其最终查询出来的结果集就是图9.2中显示的结果。

在子查询中使用IN 运算符，可以将与子查询中检索出来的结果集中的任何一个值匹配的结果都查询出来，也可以使用NOT IN 关键字明确地查询出与子查询中检索出来的结果集中不匹配的值的值的信息。例如，对于例9.2，如果要查询和教师姓名为毛翠的老师不在同一个系的教师信息，就可以使用NOT IN 关键字来完成。

**例9.3** 查询和教师姓名为毛翠的老师不在同一个系的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession,salary
FROM T_teacher
WHERE dept NOT IN
(SELECT dept
FROM T_teacher
WHERE teaName = '毛翠')
ORDER BY salary ASC
```



这里要查询和教师姓名为毛翠的老师不在同一个系的教师信息。在外层查询的WHERE子句中使用NOT IN运算符对子查询中查询出来的任意一个值进行比较，并将子查询中与查询结果不相等的记录返回。最后使用ORDER BY子句对查询的结果进行升序排序。其查询结果如图9.3所示。

teaID	teaName	age	sex	dept	profession	salary
t156354	王新	33	女	数学系	讲师	2500
t186585	孙立	48	男	物理系	讲师	3200
t181585	李慧	40	女	物理系	教授	3500
t156355	李中	55	女	数学系	教授	4200

图9.3 查询和教师姓名为毛翠的老师不在同一个系的教师信息

这里查询的结果中一共有4条记录，这4条记录的教师所在的院系都不是计算机系的，与教师姓名为毛翠的老师也不在同一个院系。

**注意** 如果在子查询中查询出的记录不只一个，则在外层查询中子查询中查询出来的结果集中的任何一个值匹配的结果都会被显示出来。

9.2.2 使用ANY运算符的子查询

ANY运算符也用于多行子查询中。ANY运算符的含义是只要与子查询中的任何一个结果值匹配，其值都会被返回。ANY运算符在使用时需要和比较运算符[=（等于）、>=（大于等于）、<=（小于等于）、>（大于）、<（小于）、!=（不等于）、<>（不等于）]放在一起使用。ANY运算符和比较运算符结合在一起表示的意义如表9.1所示。

表9.1 ANY运算符和比较运算符结合在一起表示的意义

ANY运算符和比较运算符结合	意 义
=ANY	等于查询结果中的任何一个值。相当于IN运算符
> ANY	大于查询结果中的任何一个值。即大于查询结果中最小的值
<ANY	小于查询结果中的任何一个值。即小于查询结果中最大的值
>=ANY	大于等于查询结果中的任何一个值。即大于等于查询结果中最小的值
<=ANY	小于等于查询结果中的任何一个值。即小于等于查询结果中最大的值
<>ANY（或者!=ANY）	不等于查询结果中的任何一个值

下面来看一个使用ANY运算符进行子查询的例子。

例9.4 查询其他院系的教师中工资比任意一个数学系教师的工资都高的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession,salary
FROM T_teacher
WHERE salary >ANY
(SELECT salary
FROM T_teacher
WHERE dept = '数学系')
AND dept != '数学系'
ORDER BY salary ASC
```

这里要查询的是其他院系中教师工资比任意一个数学系教师的工资都高的教师信息。所以外层查询的WHERE子句中需要有两个条件。一个限制条件是查询教师工资比任意一个数学系教师的工资都高的教师信息，这里通过使用子查询首先查询出数学系中教师的工资，然后在外层查询的WHERE子句中使用>ANY运算符查询出比任意一个数学系教师的工资都高的教师信息；另外一个限制条件是要查询的

教师信息是其他院系，也就是说查询出来的记录不包括数学系的教师，所以这里还需要加一个dept != '数学系'这样一个条件。在这段SQL语句中使用AND关键字将这两个限制条件连接起来。最后使用ORDER BY子句对查询的结果进行升序排序。其查询的结果如图9.4所示。

teaID	teaName	sex	dept	profession	salary
t105320	于波	28 男	计算机系	讲师	2800
t102225	赵伟	38 男	计算机系	副教授	3000
t186585	孙立	48 男	物理系	讲师	3200
t181585	李慧	40 女	物理系	教授	3500
t103265	张昌	43 男	计算机系	教授	3800
t106358	毛翠	50 女	计算机系	教授	4000

图9.4 查询其他院系的教师中工资比任意一个数学系教师的工资都高的教师信息

这里一共查询出6条记录。这6条记录中有4条记录是计算机系的教师信息，有2条记录是物理系的教师信息。下面来分析一下这个结果产生的过程。

(1) 这段SQL语句会执行子查询中的查询语句，这段子查询语句如下：

```
SELECT salary
FROM T_teacher
WHERE dept = '数学系'
```

这段子查询产生的结果如下所示。

```
salary
-----
2500
4200
```

可以看到这段子查询中查询出来的数学系的教师工资分别是2500元和4200元。

(2) 子查询的查询结果产生之后，在外层WHERE子句中使用>ANY运算符限制查询的结果。>ANY运算符的意义已经在表9.1中列出，其外层查询的结果要大于子查询结果中的任何一个值。而子查询的结果在第一步中已经得到，查询出来的是两个值2500和4200。根据>ANY运算符的意义，只要教师信息表中教师的工资比2500和4200这两个值中任何一个值大（即只要比2500的值大），则该教师的信息都会作为最后的查询结果显示出来。其查询的结果如下所示。

teaID	teaName	age	sex	dept	profession	salary
t102225	赵伟	38	男	计算机系	副教授	3000
t103265	张昌	43	男	计算机系	教授	3800
t105320	于波	28	男	计算机系	讲师	2800
t106358	毛翠	50	女	计算机系	教授	4000
t156355	李中	55	女	数学系	教授	4200
t181585	李慧	40	女	物理系	教授	3500
t186585	孙立	48	男	物理系	讲师	3200

7 rows in set

这个查询结果中一共有7条记录，其中有4条记录是计算机系的教师信息，有1条是数学系的教师信息，有2条记录是物理系的教师信息。这7条记录中教师的工资都比2500大。

(3) 在外层查询的WHERE子句中还有一个dept != '数学系'的限制条件，这个限定条件就把步骤2中查询出来的结果集中的数学系里教师姓名为李中的教师信息过滤掉了。

(4) ORDER BY子句放在查询语句的最后，对产生的结果按照工资由低到高升序的顺序排序。其最终查询出来的结果集就是图9.4中显示的结果。



**注意** ANY运算符不能单独使用，需要和像=（等于）、>=（大于等于）、<=（小于等于）、>（大于）、<（小于）、!=（不等于）、<>（不等于）这样的算术运算符结合在一起使用。

9.2.3 使用ALL运算符的子查询

ALL运算符也用于多行子查询中。ALL运算符的含义是与子查询中的所有的结果值匹配时，其值才会被返回。ALL运算符在使用时需要和比较运算符[=（等于）、>=（大于等于）、<=（小于等于）、>（大于）、<（小于）、!=（不等于）、<>（不等于）]放在一起使用。ALL运算符和比较运算符结合在一起表示的意义如表9.2所示。

表9.2 ALL运算符和比较运算符结合在一起表示的意义

ALL运算符和比较运算符结合	意 义
=ALL	等于查询结果中的所有的值
> ALL	大于查询结果中的所有的值。即大于查询结果中最大的值
<ALL	小于查询结果中的所有的值。即小于查询结果中最小的值
>=ALL	大于等于查询结果中的所有的值。即大于等于查询结果中最大的值
<=ALL	小于等于查询结果中的所有的值。即小于等于查询结果中最小的值
<>ALL（或者!= ALL）	不等于查询结果中的所有的值。相当于NOT IN运算符

下面来看一个使用ALL运算符进行子查询的例子。

例9.5 查询其他院系的教师中工资比物理系工资最少的教师还低的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession,salary
FROM T_teacher
WHERE salary <ALL
(SELECT salary
FROM T_teacher
WHERE dept = '物理系')
AND dept != '物理系'
```

这里要查询的是其他院系的教师中工资比物理系工资最少的教师还低的教师信息。所以外层查询的WHERE子句中需要有两个条件。一个限制条件是查询教师工资比物理系工资最少的教师还低的教师信息，这里通过使用子查询首先查询出物理系中教师的工资，然后在外层查询的WHERE子句中使用<ALL运算符查询出比物理系教师工资最少的教师工资还低的教师信息；另外一个限制条件是要查询的教师信息是其他院系，也就是说查询出来的记录不包括物理系的教师，所以这里还需要加一个dept != '物理系'这样一个条件。在这段SQL语句中使用AND关键字将这两个限制条件连接起来。最后使用ORDER BY子句对查询的结果进行升序排序。其查询的结果如图9.5所示。

teaID	teaName	age	sex	dept	profession	salary
t156354	王新	33	女	数学系	讲师	2500
t105320	于波	26	男	计算机系	讲师	2800
t102225	赵伟	38	男	计算机系	副教授	3000

图9.5 查询其他院系的教师中工资比物理系工资最少的教师还低的教师信息

这里一共查询出3条记录。这3条记录中有2条记录是计算机系的教师信息，有1条记录是物理系的教师信息。下面来分析一下这个结果产生的过程。

(1) 这段SQL语句会执行子查询中的查询语句，这段子查询语句如下：

```
SELECT salary
FROM T_teacher
WHERE dept = '物理系'
```

这段子查询产生的结果如下所示。

```
salary
-----
3500
3200
```

可以看到这段子查询中查询出来的物理系的教师工资分别是3500元和3200元。

(2) 子查询的查询结果产生之后，在外部WHERE子句中使用<ALL运算符限制查询的结果。<ALL运算符的意义已经在表9.2中列出，其外层查询的结果要大于子查询结果中的所有的值。而子查询的结果在第一步中已经得到，查询出来的是两个值3500和3200。根据<ALL运算符意义，教师信息表中教师的工资需要比3500和3200这两个值都小（即需要比3200的值小），则该教师的信息都会作为最后的查询结果显示出来。其查询的结果如下所示。

teaID	teaName	age	sex	dept	profession	salary
t102225	赵伟	38	男	计算机系	副教授	3000
t105320	于波	28	男	计算机系	讲师	2800
t156354	王新	33	女	数学系	讲师	2500

3 rows in set

这个查询结果中一共有3条记录，其中有2条记录是计算机系的教师信息，有1条是数学系的教师信息，这3条记录中教师的工资都比3200要小。

(3) 在外层查询的WHERE子句中还有一个dept != '物理系'的限定条件，这个限定条件就把步骤2中查询出来的结果集中的物理系里教师信息过滤掉了。当然，这里没有查到物理系的教师。

(4) ORDER BY子句放在查询语句的最后，对产生的结果按照工资由低到高升序的顺序排序。其最终查询出来的结果集就是图9.5中显示的结果。

**注意** ALL运算符不能单独使用，需要和像=（等于）、>=（大于等于）、<=（小于等于）、>（大于）、<（小于）、!=（不等于）、<>（不等于）这样的算术运算符结合在一起使用。

### 9.3 多列子查询

所谓多列子查询，是指子查询的语句会返回多个数据列的子查询语句。在WHERE子句中也可以使用将多个属性值用括号括起来的方式实现多列子查询。在多列子查询中，WHERE子句中需要使用括号将多个属性括在一起，多个属性之间需要用逗号分开。

外层查询的WHERE子句中根据多列子查询返回的行数不同，可以选择使用不同的运算符。如果多列子查询中返回的数据行是单行的，即返回的结果值只有一个，则可以使用算术比较运算符；如果子查询中返回的数据行是多行的，即返回的结果值不只一个，则可以使用IN、ANY、ALL运算符。

### 例9.6 查询院系和职称都与教师编号为t103265教师相同的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
WHERE (dept,profession) =
(SELECT dept,profession
FROM T_teacher
WHERE teaID = 't103265')
```

这里要查询的是院系和职称都与教师编号为t103265教师相同的教师信息，这里要比较的属性有两个，一个是教师所在的院系，另一个是教师的职称。在子查询中首先需要将编号为t103265教师的所在的院系和该教师的职称查询出来，这里可以很明确地知道该子查询中返回的结果是一个值。由于比较的属性有两个，在外部的WHERE子句中需要将这两个属性用括号括起来，然后使用等于(=)运算符将与教师编号为t103265教师所在院系和职称相同的教师信息查询出来。其查询结果如图9.6所示。

teaID	teaName	age	sex	dept	profession
t103265	张昌	43	男	计算机系	教授
t106358	毛翠	50	女	计算机系	教授

图9.6 查询院系和职称都与教师编号为t103265教师相同的教师信息

**注意** 在多列子查询中，如果比较的多个列是成对比较，则要求外层查询的WHERE子句中数据列属性的数据类型要和子查询中返回的查询结果的数据列的数据类型必须是同时匹配的。如果不同时匹配，查询可能就会出现错误。

这里查询的结果中一共有2条记录，这2条记录的教师所在的院系都是计算机系，教师职称都是教授。下面来分析一下这个结果产生的过程。

(1) 这段SQL语句首先执行的是子查询中的语句，在例9.6中子查询的SQL语句如下：

```
SELECT dept,profession
FROM T_teacher
WHERE teaID = 't103265'
```

这条子查询语句查询的是教师编号为t103265教师所在的院系和职称。其查询结果如下所示。

```
dept      profession
-----
计算机系  教授
```

这里的子查询中查询出教师编号为t103265教师所在的院系是计算机系，职称为教授。

(2) 将子查询的结果作为外层查询的查询条件。子查询中返回的结果包含了两个数据列，分别是表示教师所在院系的数据列dept和表示教师职称的数据列profession。因此在外层查询的WHERE子句中需要比的属性就应该有两个：dept和profession。并用括号将这两个属性括起来，两个属性之间用逗号分开。

(3) 通过使用运算符(=)对dept和profession这两个属性值进行比较，并将比较相等的结果（即教师院系在计算机系，职称为教授的教师）作为结果集返回。其最终查询出来的结果集就是图9.6中显示的结果。

现在请读者思考一个问题，如果现在要教师信息表中与教师姓名为张昌的教师所在院系和职称都相等的教师信息，在外层查询的WHERE子句中应该使用等号运算符(=)还是IN运算符呢？

**提示** 教师信息表中姓名为张昌的教师可能不只一个，如果有两个或者多个教师的姓名相同的情况，则其子查询的SQL语句可能会返回多个值。因此为了保证查询结果的正确执行，最好还是在外层查询的WHERE子句中使用IN运算符。

其实，多列子查询在实际应用中使用的并不是很多，很多情况下，通过使用其他的SQL语句也完全可以达到和多列子查询完全相同的查询效果。读者可以考虑一下如果使用其他的方法实现例9.6中的查询结果（可以参考9.6节）。

## 9.4 相关子查询

在前面介绍的SQL语句子查询中，都是首先执行内层子查询的语句，然后将子查询返回的结果作为外层查询的查询条件检索数据的。这时的子查询只执行一次。而相关子查询中，子查询需要重复执行。每处理一行外部的查询语句，子查询都会被执行一次。也就是说，相关子查询需要依赖于外层查询，外层查询和子查询之间存在联系。可以使用EXISTS关键字或者NOT EXISTS关键字实现相关子查询。这一节主要介绍带有EXISTS关键字的相关子查询和带有NOT EXISTS关键字的相关子查询。

### 9.4.1 带有EXISTS关键字的相关子查询

在SQL语句中，可以使用带有EXISTS关键字的子查询实现相关子查询的操作。带有EXISTS关键字的子查询在执行时只会返回逻辑值TRUE或者FALSE，而不会返回任何数据。也就是说，带有EXISTS关键字的子查询不关心返回的是什么数据，而只关心返回的数据“有还是没有”。

如果EXISTS子句的子查询中有返回的结果，则外层查询的WHERE子句就返回TRUE，则该结果就会作为最终查询的结果集显示出来；如果EXISTS子句的子查询中没有返回的结果，则外层查询的WHERE子句就返回FALSE。这个过程会反复执行，直到将外层查询的数据全部检查完毕为止。下面通过一个例子来看一下如何实现带有EXISTS关键字的子查询。

**例9.7** 查询选修课程编号为t105这门课的学生信息。

```
SELECT S.stuID ,S.stuName,S.age,S.sex
FROM T_student S
WHERE EXISTS
(SELECT *
FROM T_result T
WHERE
S.stuID=T.stuID
AND T.curID = 't105')
```

这个SQL语句使用了带有EXISTS关键字的子查询的方法检索选修课程编号为t105这门课的学生信息。其查询的结果如图9.7所示。

这里查询的结果中一共有3条记录，分别显示了选修课程编号为t105这门课的3个学生的信息。下面来分析一下这个结果产生的过程。这个处理过程中需要用到两张表：学生信息表（T\_student）和成绩信息表（T\_result）。这两张数据表结构如下所示。

学生信息表（T\_student）：

stuID	stuName	age	sex
s102203	赵亮	23	男
s206363	张明	22	男
s253263	李凤	24	女

图9.7 查询选修课程编号为t105这门课的学生信息

## 零基础学SQL

stuID	stuName	age	sex	birth
s102203	赵亮	23	男	1986-05-16 00:00:00
s112303	郑茹	21	女	1988-01-25 00:00:00
s115263	王海	23	男	1986-08-02 00:00:00
s206363	张明	22	男	1987-04-23 00:00:00
s221256	王昌鹤	24	男	1985-03-18 00:00:00
s231456	王玉梅	22	女	1987-03-28 00:00:00
s232516	李玉峰	24	女	1985-08-16 00:00:00
s253263	李凤	24	女	1985-06-06 00:00:00

8 rows in set

成绩信息表 (T\_result):

stuID	curID	result
s102203	t105	85
s102203	t232	75
s102203	t321	90
s102203	t333	60
s112303	t321	65
s112303	t333	75
s206363	t105	80
s206363	t333	55
s221256	t232	60
s253263	232	70
s253263	t105	50
s253263	t321	90
s253263	t333	53

13 rows in set

(1) 外层查询中首先取得学生信息表 (T\_student) 中的第一个学生的信息，第一个学生的学生编号为s102203。

(2) 在子查询中取得学生编号为s102203这条记录，在成绩信息表 (T\_result) 中查询，查询是否有编号为s102203这名学生，然后还要判断学生是否选取了课程编号为t105这门课。在成绩信息表 (T\_result) 中可以看到学生编号为s102203这名学生选择了课程编号为t105这门课，因此在子查询中会返回一个值TRUE。

(3) WHERE子句中带有EXISTS关键字的子查询返回值是TRUE，学生编号为s102203这条学生信息就作为查询结果集中的一条记录显示出来。

(4) 外层查询中再取得学生信息表中的第二个学生的信息，第二个学生的学生编号为s112303。

(5) 在子查询中取得学生编号为s112303这条记录，在成绩信息表 (T\_result) 中查询，查询是否有编号为s112303这名学生，并判断该学生是否选取了课程编号为t105这门课。在成绩信息表 (T\_result) 中可以看到学生编号为s112303这名学生没有选择课程编号为t105这门课，因此在子查询中会返回一个



值FALSE。

(6) WHERE子句中带有EXISTS关键字的子查询返回值是FALSE，学生编号为s112303这条学生信息就不会显示出来。

(7) 按照以上的查询方式依次类推，直到将学生信息表（T\_student）中所有学生的信息全部检查完毕为止。最后查询的结果集就是图9.7所显示的结果。

**注意** 在使用EXISTS关键字进行相关子查询时，EXISTS子句的子查询里的SELECT语句选择的列一般用\*代替。因为子查询中只关心是否会返回结果，并不关心返回的值到底是什么。因此即使在SELECT语句中给出要查询列的名字，也是没有意义的。

例9.7中使用的是带有EXISTS关键字的子查询实现相关子查询的方法实现了查询选修课程编号为t105这门课的学生信息，其实使用连接查询的方法也可以实现例9.7。读者可以思考一下使用连接查询的方法如何实现例9.7。

**说明** 由于相关子查询中，子查询需要重复执行，在查询数据时，数据库管理系统的负担也会比较大，如果在子查询中嵌套层数过多，也会影响到查询的效率，因此如果SQL语句中需要嵌套的子查询层数过多，可以考虑将其修改为连接查询。

#### 9.4.2 带有NOT EXISTS关键字的相关子查询

有EXISTS关键字实现相关子查询，自然也就可以使用NOT EXISTS关键字实行相关子查询。使用NOT EXISTS关键字实行相关子查询的查询方法和使用EXISTS关键字实行相关子查询的查询方法正好相反。如果NOT EXISTS子句的子查询中没有返回的结果，则外层查询的WHERE子句就返回TRUE，则此最终查询的结果集显示出来；如果NOT EXISTS子句的子查询中有返回的结果，则外层查询的WHERE子句就返回FALSE。下面通过一个例子来看一下如何实现带有NOT EXISTS关键字的子查询。

**例9.8** 查询没有选修课程编号为t105这门课的学生信息（使用NOT EXISTS）。

```
SELECT S.stuID ,S.stuName,S.age,S.sex
FROM T_student S
WHERE NOT EXISTS
(SELECT *
FROM T_result T
WHERE
S.stuID=T.stuID
AND T.curID = 't105')
```

这个SQL语句使用了带有NOT EXISTS关键字的子查询的方法检索选修课程编号为t105这门课的学生信息。其查询的结果如图9.8所示。

这里查询的结果中一共有5条记录，分别显示了没有选修课程编号为t105这门课的5个学生的信息。可以看到其查询的结果正好和图9.7显示的结果相反。下面来分析一下这个结果产生的过程。

stuID	stuName	age	sex
s112303	郑茹	21	女
s115263	王海	23	男
s221256	王昌群	24	男
s231456	王玉梅	22	女
s232516	李玉梅	24	女

图9.8 查询没有选修课程编号为t105这门课的学生信息（使用NOT EXISTS）

(1) 外层查询中首先取得学生信息表（T\_student）中的第一个学生的信息，第一个学生的学生编

号为s102203。

(2) 在子查询中取得学生编号为s102203这条记录，在成绩信息表（T\_result）中查询，查询是否有编号为s102203这名学生，然后还要判断学生是否选取了课程编号为t105这门课。在成绩信息表（T\_result）中可以看到学生编号为s102203这名学生选择了课程编号为t105这门课，因此在子查询中会返回一个值TRUE。

(3) WHERE子句中带有NOT EXISTS关键字的子查询返回值是FALSE，学生编号为s102203这条学生信息就不会显示出来。

(4) 外层查询中再取得学生信息表中的第二个学生的信息，第二个学生的学生编号为s112303。

(5) 在子查询中取得学生编号为s112303这条记录，在成绩信息表（T\_result）中查询，查询是否有编号为s112303这名学生，并判断该学生是否选取了课程编号为t105这门课。在成绩信息表（T\_result）中可以看到学生编号为s112303这名学生没有选择课程编号为t105这门课，因此在子查询中会返回一个值FALSE。

(6) WHERE子句中带有NOT EXISTS关键字的子查询返回值是TRUE，学生编号为s112303这条学生信息就作为查询结果集中的一条记录显示出来。

(7) 按照以上的查询方式依次类推，直到将学生信息表（T\_student）中所有学生的信息全部检查完毕为止。最后查询的结果集就是图9.8所显示的结果。

**注意**

在使用NOT EXISTS关键字进行相关子查询时，NOT EXISTS子句的子查询里的SELECT语句选择的列也都需要用\*代替。

**说明**

在相关子查询中，外部查询和子查询是有联系的。子查询的查询条件要依赖于外层查询中的属性值。

## 9.5 在SQL语句中使用子查询

子查询语句除了可以应用在WHERE子句中，也可以应用在SELECT子句、FROM子句、ORDER BY子句、HAVING子句、CREATE TABLE 语句、CREATE VIEW 语句、INSERT 语句、UPDATE语句、DELETE等语句中。本节主要介绍子查询在SELECT子句、FROM子句和HAVING子句中的应用。有关子查询在CREATE TABLE 语句中的应用可以参看9.7节，有关子查询在CREATE VIEW语句中的应用可以参看第11章；有关子查询在INSERT语句、UPDATE语句、DELETE语句的应用可以参看第12章～第14章。

### 9.5.1 在SELECT子句中使用子查询

在SELECT子句中使用子查询，该子查询查询出来的结果应该是一个具体的值。下面来看一个在SELECT子句中使用子查询的例子。

**例9.9** 查询学生编号为s102203的成绩信息（在SELECT子句中使用子查询）。

```
SELECT R.stuID,  
(SELECT stuName  
FROM T_student  
WHERE stuID = R.stuID) AS stuName, R.result, R.curID
```

```
FROM T_result R
WHERE R.stuID = 's102203'
ORDER BY R.result ASC
```

这里在SELECT语句中使用了子查询，在子查询中使用WHERE子句限定查询条件，查询学生编号为s102203对应的学生姓名。子查询用括号括起来，在该子查询之后还需要为其指定一个别名，这里指定子查询的别名为stuName。最后使用ORDER BY子句对成绩进行升序排序。其查询结果如图9.9所示。

stuID	stuName	curID	result
s102203	赵亮	t333	60
s102203	赵亮	t232	75
s102203	赵亮	t105	85
s102203	赵亮	t321	90

图9.9 使用子查询查询学生编号为s102203的成绩信息

**注意** 在SELECT子句中使用子查询时，子查询查询出来的结果应该是一个具体的值。

9.5.2 在FROM子句中使用子查询

在FROM子句中可以使用子查询，该子查询查询出来的结果集组成一个临时的数据表。下面来看一个在FROM子句中使用子查询的例子。

例9.10 查询学生编号为s102203学生的选课成绩信息（在FROM子句中使用子查询）。

```
SELECT R.stuID,C.curID, C.curName,R.result
FROM T_curriculum C,
(SELECT curID,stuID,result
FROM T_result)R
WHERE R.curID=C.curID
AND R.stuID = 's102203'
ORDER BY R.result ASC
```

这里在FROM子句中使用了子查询，查询成绩信息表(T\_result)中课程编号、学生编号和课程的成绩，子查询用括号括起来，在该子查询之后还需要为其指定一个别名，这里的别名为R。最后使用ORDER BY子句对成绩进行升序排序。其查询结果如图9.10所示。

stuID	curID	curName	result
s102203	t333	高等数学	60
s102203	t232	数据库基础	75
s102203	t105	计算机系统结构	85
s102203	t321	C语言	90

图9.10 查询学生编号为s102203学生的选课成绩信息

**注意** 在FROM子句中使用子查询时，要指定子查询的别名。

这里一共查询出了4条记录。这4条记录都表示的是学生编号为s102203的选修课课程与成绩。下面来分析一下这个结果产生的过程。

(1) 这段SQL语句会执行FROM语句中子查询的语句，在例9.10中子查询的SQL语句如下：

```
SELECT curID,stuID,result
FROM T_result
```

这条语句查询成绩信息表(T\_result)中学生编号、课程编号和课程成绩信息。其查询结果如下所示。

curID	stuID	result
t105	s102203	85
t232	s102203	75

t321	s102203	90
t333	s102203	60
t321	s112303	65
t333	s112303	75
t105	s206363	80
t333	s206363	55
t232	s221256	60
t232	s253263	70
t105	s253263	50
t321	s253263	90
t333	s253263	53

-----+  
13 rows in set

在成绩信息表（T\_result）共有13条记录。

(2) 将该子查询返回的结果集组成一个临时数据表作为外层查询SELECT语句的查询对象，在WHERE子句中将课程信息表（T\_curriculum）和由子查询临时创建出来的表通过课程编号curID这个字段连接起来，并加上R.stuID = 's102203'的限制条件，将学生编号为s102203学生的选课成绩信息查询出来。

(3) 使用ORDER BY子句对查询出来的结果集按照成绩由低到高的顺序升序排序。其最终查询出来的结果集就是图9.10中显示的结果。

### 9.5.3 在HAVING子句中使用子查询

在HAVING子句中可以使用子查询，该子查询查询出来的结果集组成一个临时的数据表。下面来看一个在HAVING子句中使用子查询的例子。

**例9.11** 查询以学生编号s2开头的学生的平均成绩。

```
SELECT R.stuID, AVG(R.result)
FROM T_result R,T_curriculum C
WHERE R.curID=C.curID
GROUP BY R.stuID
HAVING R.stuID IN
(SELECT stuID
FROM T_student
WHERE stuID LIKE 's2%')
ORDER BY R.stuID
```

这里在HAVING子句中使用了子查询，用来查询学生编号s2开头的学生信息，并将查询到的学生信息作为分组条件。子查询用括号括起来，最后使用ORDER BY子句对成绩进行升序排序。其查询结果如图9.11所示。

这里一共查询出了3条记录。这3条记录的学生编号都是以s2开头的。下面来分析一下这个结果产生的过程。

(1) 这段SQL语句的外层查询是用来查询成绩信息表（T\_result）和课程信息表（T\_curriculum）中学生选课的课程平均成绩。

stuID	AVG(R.result)
s206363	67.5
s221256	60
s253263	65.75

图9.11 查询以学生编号s2开头的学生的平均成绩

```
SELECT R.stuID, AVG(R.result)
FROM T_result R,T_curriculum C
WHERE R.curID=C.curID
GROUP BY R.stuID
```

其查询结果如下所示。

```
+-----+-----+-----+
| stuID |
+-----+-----+-----+
| s102203 |
| s111111 |
| s206363 |
| s221256 |
| s253263 |
+-----+-----+-----+
5 rows in set
```

(2) HAVING子句中子查询用来查询学生编号s2开头的学生信息。

```
SELECT stuID
FROM T_student
WHERE stuID LIKE 's2%'
```

其查询结果如下所示。

```
+-----+-----+
| stuID |
+-----+-----+
| s206363 |
| s221256 |
| s231456 |
| s232516 |
| s253263 |
+-----+-----+
5 rows in set
```

在学生信息表（T\_ student）中共查询出5条记录。HAVING子句中使用IN运算符将子查询返回的结果集中与外层查询中学生编号相匹配的结果检索出来，并将其作为分组条件。

(3) 使用ORDER BY子句对查询出来的结果集按照成绩由低到高的顺序升序排序。其最终查询出来的结果集就是图9.11中显示的结果。

## 9.6 多重子查询

多重子查询允许查询条件中有多个子查询语句。例如在例9.6中，就可以使用多重子查询的方式来实现。其SQL语句如下：

```
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
WHERE dept =
(SELECT dept
```



## 零基础学SQL

```
FROM T_teacher
WHERE teaID = 't103265')
AND
profession =
(SELECT profession
FROM T_teacher
WHERE teaID = 't103265')
```

这个SQL语句在WHERE子句中需要两个限制条件，一个限制条件是要查询与教师编号为t103265教师所在院系相同的院系信息，另一个限制条件是要查询与教师编号为t103265教师职称相同的职称信息，然后将这两个限制条件使用AND关键字连接起来。

这个SQL语句的查询结果和使用多列子查询查询出的结果是相同的，但是可以看到，这个SQL语句的书写要比多列子查询的烦琐。在SQL语句中，通过使用多列子查询可以简化SQL语句的查询方法。但是在是使用多列子查询时，成对比较查询是要求外层查询的WHERE子句中数据列属性的数据类型要和子查询中返回的查询结果的数据列的数据类型必须是同时匹配的。例如下面这个例子就只能用多重子查询的方式来实现。

**例9.12** 查询教师信息表中职称与教师编号为t181585教师相同但工资比该教师高的教师信息。

```
SELECT teaID,teaName,age,sex,dept,profession,salary
FROM T_teacher
WHERE profession =
(SELECT profession
FROM T_teacher
WHERE teaID = 't181585')
AND
salary>
(SELECT salary
FROM T_teacher
WHERE teaID = 't181585')
```

这个SQL语句在WHERE子句中需要两个限制条件，一个限制条件是要查询教师编号为t181585教师职称相同的职称信息，另一个限制条件是要查询比教师编号为t181585工资高的教师的工资信息，然后将这两个限制条件使用AND关键字连接起来。其查询的结果如图9.12所示。

这里查询的结果中一共有3条记录，这3条记录中有2条记录是计算机系的教师，有1条记录是数学系教师的信息，这些教师职称都是教授。下面来分析一下这个结果产生的过程。

(1) 这段SQL语句中有两个子查询。第一个子查询是用来查询教师编号为t181585教师的职称信息的。其SQL语句如下：

```
SELECT profession
FROM T_teacher
WHERE teaID = 't181585'
```

其查询结果如下所示。

teaID	teaName	age	sex	dept	profession	salary
t103265	张晶	43	男	计算机系	教授	3800
t106358	毛翠	50	女	计算机系	教授	4000
t156395	李中	55	女	数学系	教授	4200

图9.12 查询教师信息表中职称与教师编号为t181585教师相同但工资比该教师高的教师信息

```
profession
```

```
-----  
教授
```

这里的子查询中查询出教师编号为t181585教师的职称为教授。

(2) 第二个子查询是用来查询教师编号为t181585教师的工资信息的。其SQL语句如下：

```
SELECT salary  
FROM T_teacher  
WHERE teaID = 't181585'
```

其查询结果如下所示。

```
salary
```

```
-----  
3500
```

这里的子查询中查询出教师编号为t181585教师工资是3500元。

(3) 将子查询中查询到的结果作为外层查询的查询条件。在外层查询的WHERE子句中分别对这两个子查询返回的结果进行比较。使用等号运算符(=)让表示教师职称的数据列profession和子查询中查询出来的结果进行比较，使用大于运算符(>)与子查询中查询出来的工资进行比较，并用AND关键字将这两个条件连接起来。其最终查询出来的结果集就是图9.12中显示的结果。

SQL语句中除了允许查询条件中有多个子查询语句，还允许在SELECT语句中实现多层的嵌套。Oracle数据库中最多允许子查询嵌套16层。但是过多的子查询的嵌套会不利于开发人员或者用户对SQL语句的理解。

#### 提示

在SQL语句中允许查询条件中有多个子查询语句，如果在嵌套含有多个子查询语句中存在错误，可能会让人产生迷惑，不知从哪儿入手进行调试。对于嵌套含有多个子查询的SQL语句，可以采用逐步求值验证的方法进行调试。即首先运行最内层的子查询语句，如果最内层的子查询语句没有问题，再运行其外层的子查询语句，以此类推，如果嵌套的子查询语句都没有问题，就将它们再重新放入主查询语句中，这样从内而外，逐步调试。

子查询中嵌套的层次过多，会影响查询效率。应该尽量避免过多的子查询的嵌套。如果在SQL语句中需要使用子查询的嵌套，应该在子查询中尽量将无用的数据过滤掉。

## 9.7 在CREATE TABLE语句中使用子查询实现数据表的复制

子查询可以应用在CREATE TABLE语句中，通过在CREATE TABLE语句中使用子查询可以在建立一张新的数据表的同时将原有表中的数据插入到新建的数据表中，即实现数据表中数据的复制功能。CREATE TABLE语句使用子查询的语法格式如下：

```
CREATE TABLE 表名  
AS  
SELECT 语句
```

其中，CREATE TABLE是创建表的关键字，表名表示要创建新的数据表的名字，AS关键字后面跟的就是创建表的子查询语句。下面来看一个CREATE TABLE语句中使用子查询的例子。

例9.13 创建一张新的学生信息表，要求新表中包括原来学生信息表（T\_student）的所有数据。

```
CREATE TABLE T2_student
AS
SELECT stuID,stuName,age,sex,birth
FROM T_student
```

这里，T2\_student表示要创建的新表，数据库会根据AS关键字后面跟的子查询语句SELECT语句中所选择的列的数据全部复制到新建的数据表中。此时，使用SELECT语句查询T2\_student表中的数据，可以看到查询结果如图9.13所示。

stuID	stuName	age	sex	birth
s102203	赵亮	23	男	1986-05-16 00:00:00
s112303	郑茜	21	女	1988-01-25 00:00:00
s115263	王海	23	男	1986-08-02 00:00:00
s206363	张明	22	男	1987-04-23 00:00:00
s221256	王昌辉	24	男	1985-03-19 00:00:00
s231456	王玉梅	22	女	1987-03-28 00:00:00
s232516	李玉峰	24	女	1985-08-16 00:00:00
s253263	李凤	24	女	1985-06-06 00:00:00

图9.13 T2\_student表的查询结果

图9.13中显示的查询结果和原来学生信息表（T\_student）中的结果是一样的，即通过在CREATE TABLE语句中使用子查询的方式实现了数据表的复制。但是这并不是一种非常好的复制方式，因为在CREATE TABLE语句中使用子查询的方式实现数据表的复制过程中，新建的数据表T2\_student中的索引以及数据列的一些属性会发生一些变化。在MySQL Command Line Client中可以看到这两张表在创建过程中的不同。

单击“开始”|“所有程序”|“MySQL”|“MySQL Server 5.0”|“MySQL Command Line Client”命令，输入密码，并使用“USE 数据库名”命令选择使用的数据库名，然后在MySQL Command Line Client窗口中输入如下命令：

```
SHOW CREATE TABLE T_student;
```

在MySQL Command Line Client窗口中输入该命令后，按“Enter”键，在屏幕上会显示T\_student表的创建语句如下所示。

```
+-----+
| T_student | CREATE TABLE 't_student' (
| 'stuID' varchar(15) NOT NULL,
| 'stuName' varchar(10) character set gb2312 NOT NULL,
| 'age' int(11) NOT NULL,
| 'sex' varchar(2) character set gb2312 NOT NULL,
| 'birth' datetime NOT NULL,
| PRIMARY KEY ('stuID')
| ) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+
```

在MySQL Command Line Client中，再输入以下命令：

```
SHOW CREATE TABLE T2_student;
```

在MySQL Command Line Client窗口中输入该命令后，按“Enter”键，在屏幕上会显示T2\_student表的创建语句如下所示。

```
+-----+
| T2_student | CREATE TABLE 't2_student' (
| 'stuID' varchar(15) NOT NULL default '',
| 'stuName' varchar(10) character set gb2312 NOT NULL default '',
| 'age' int(11) NOT NULL default '0',
| 'sex' varchar(2) character set gb2312 NOT NULL default '',
+-----+
```

```
'birth' datetim NOT NULL default '0000-00-00 00:00:00'
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-----+
```

通过这两张表的创建过程，可以比较它们之间的不同。在CREATE TABLE语句中使用子查询实现数据表的复制时，新创建的数据表中不能继承原数据表中的主键、约束条件等信息，新创建的数据表的主键、约束条件的设置需要自己手工完成。

除了使用这种方法复制表中的数据以外，还可以在INSERT INTO语句中使用子查询的方法实现数据表的复制。这部分内容可以参考12.1.3节。

当然对于每一个数据库来说，使用数据库本身提供的数据表的导入/导出功能也可以实现数据表之间的复制。关于这部分内容读者可以参考有关专门介绍数据库操作和使用的书籍。这部分内容已经超出了本书所讲的范围。

**说明** 在CREATE TABLE语句中使用子查询建立新表时，可以不必指定要创建的新表的列。MySQL数据库会根据SELECT语句中所选择的列的数据在新表中自动创建相应的列。

## 9.8 小结

本章主要介绍了不同子查询的作用和使用方法。根据子查询返回的行数不同，可以分为返回一个值的子查询和返回多行值的子查询。如果子查询返回一个值，可以使用单行的比较运算符，如果子查询返回的是多行值，就需要使用IN、ANY、ALL等运算符。在成对比较的多列子查询中，要注意外层查询的WHERE子句中数据列属性的数据类型要和子查询中返回的查询结果的数据列的数据类型同时匹配。

根据返回的数据与外层查询之间的关系，可以分为相关子查询和不相关子查询。在相关子查询中需要掌握使用EXISTS关键字和NOT EXISTS关键字实现相关子查询的查询过程。最后需要了解，子查询除了可以应用在WHERE子句中，也可以应用在FROM语句以及表的创建语句中。

虽然子查询可以多层嵌套，但是一般情况下嵌套的层数不易过多，过多的嵌套会导致SQL语句理解上的困难。子查询还可以应用在INSERT语句、UPDATE语句和DELETE语句这些数据操纵语言中，有关子查询在数据操纵语言中的应用可以参考第四篇中的讲解。

## 第10章 常用函数

在实际应用中，经常需要对字符串、数字以及日期时间等数据进行处理。例如，字符串的截取、字符串的拼接、字符串的替换等；求指定数字的幂、三角函数、四舍五入操作等；取得当前系统日期、对日期进行加减运算等，这些处理都需要使用数据库中提供的内置函数来完成。

在本章的大部分的例子中，都需要用到一张表——dual表。所有的用户都可以访问dual表，该表只包含一条记录。它是一张很有用的表，在使用SELECT语句对函数进行测试时，会经常用到这张表。本章在讲解函数的使用方法时，也会用到这张表。

不同的数据库中，都提供了有关对字符串、数字以及日期时间等数据进行处理的功能。本章主要以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，讲解在这3种数据库中一些常用函数的功能及其使用方法。

本章重点：

- ☐ 字符函数
- ☐ 数字函数
- ☐ 日期时间函数
- ☐ 转换函数
- ☐ 比较函数
- ☐ 空值处理函数
- ☐ 分支函数和条件表达式

### 10.1 字符函数

字符函数中要求输入的参数是字符类型的值，其返回值可以是一个数字类型的值也可以是一个字符类型的值。字符函数有很多，不同的数据库中完成相同功能的字符函数也不完全相同。本节将介绍一些在实际中经常用到的字符函数。这里以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，对于一些字符函数在这几种数据库中的一些不同的使用方法都做了比较详细的介绍。

#### 10.1.1 计算字符串长度

在SQL语句中，如果希望取得一个指定的字符串的长度，可以使用LENGTH函数。该函数的语法规则如下：

```
LENGTH(string)
```

该函数的功能是计算字符串长度。其中，参数string表示指定要计算字符串长度的字符串。如果字符串类型为CHAR，则LENGTH函数返回的值包括该字符串中所含有的空格。



### 例10.1 计算给定字符串的长度。

```
SELECT LENGTH(' CREATE TABLE ')
FROM dual
```

这段SQL语句是计算字符串CREATE TABLE的长度。这个字符串中包含了3个空格，单词CREATE前面有一个空格，单词CREATE 和TABLE之间有一个空格，单词TABLE之后还有个空格。其查询结果如下所示。

```
LENGTH('CREATE TABLE ')
-----
14
```

LENGTH函数返回的值是14。可以看到这个结果实际上将字符串CREATE TABLE中包含的3个空格也计算在内了。

**注意** 在Microsoft SQL Server数据库中，计算字符串的长度可以使用LEN函数。

MySQL数据库中，如果FROM子句中使用的是dual表，则FROM子句可以省略，可以直接写成下面的这种形式：

```
SELECT 函数(参数1[,参数2,...]);
```

例如例10.1，就可以使用下面的SQL语句完成。

```
SELECT LENGTH(' CREATE TABLE ');
```

这段SQL语句也是取得字符串CREATE TABLE的长度。这里在SELECT语句之后多了一个分号并省略了FROM子句，其查询的结果不变。

需要指出的是，在MySQL数据库中，使用LENGTH函数计算字符串的长度时，如果字符串中的字符是数字或者是字母，则字符串的长度按照一个字符来计算，如果字符串中含有汉字，则一个汉字相当于三个字符的长度。例如下面这个例子。

### 例10.2 计算给定字符串中含有汉字的长度。

```
SELECT LENGTH(' CREATE 教师表')
FROM dual
```

这里指定的字符串中包含有汉字，按照刚才的讲述，一个汉字要按照3个字符的长度来计算，那么该指定的字符串中的3个汉字就相当于9个字符的长度。其查询的结果如下所示。

```
LENGTH(' CREATE 教师表')
-----
17
```

LENGTH函数返回的值是17。这里单词CREATE计算长度为6，教师表这3个汉字计算的长度为9，在单词CREATE前后有两个空格，所以该字符串的长度应该是6+9+2=17。

MySQL数据库中，使用LENGTH函数计算字符串长度，一个数字或者字母按照一个字符来计算，**注意** 一个汉字按照三个字符来计算。如果希望将汉字也按照一个字符来计算，可以使用CHAR\_LENGTH函数。

## 零基础学SQL

### 10.1.2 将字符串全部转换为小写

在SQL语句中，如果希望将一个指定的字符串全部转换为小写，可以使用LOWER函数。该函数的语法格式如下：

```
LOWER(string)
```

该函数的功能是将字符串中的字母全部转换为小写的格式。其中，参数string表示指定要转换为小写字母格式的字符串。

**例10.3** 将指定的字符串全部转换为小写格式。

```
SELECT LOWER('CREATE TABLE')  
FROM dual
```

这段SQL语句是将字符串CREATE TABLE中的所有的大写字母都转换为小写的格式。其查询的结果如下所示。

```
LOWER('CREATE TABLE')  
-----  
create table
```

这里的查询结果为create table。可以看到，字符串create table是将SELECT语句中指定的字符串CREATE TABLE的每一个字母都变成了小写的结果。

### 10.1.3 将字符串全部转换为大写

在SQL语句中，如果希望将一个指定的字符串全部转换为大写，可以使用UPPER函数。该函数的语法格式如下：

```
UPPER(string)
```

该函数的功能是将字符串中的字母全部转换为大写的格式。其中，参数string表示指定要转换为大写字母格式的字符串。

**例10.4** 将指定的字符串全部转换为大写格式。

```
SELECT UPPER('create table')  
FROM dual
```

这段SQL语句是将字符串create table中的所有的小写字母都转换为大写的格式。其查询的结果如下所示。

```
UPPER('create table')  
-----  
CREATE TABLE
```

这里的查询结果为CREATE TABLE。可以看到，字符串CREATE TABLE是将SELECT语句中指定的字符串create table的每一个字母都变成了大写的结果。

### 10.1.4 将字符串中单词的首字母转换为大写

在SQL语句中，如果希望将一个指定的字符串单词的首字母转换为大写，在Oracle数据库中可以使用INITCAP函数。该函数的语法格式如下：

INITCAP(string)

该函数的功能是将字符串中每一个单词的首字母转换为大写的格式。其中，参数string表示指定要转换格式字符串。如果字符串string的值为NULL，则返回的结果为NULL。

例10.5 将指定字符串中的单词首字母变为大写。

```
SELECT INITCAP('create table' )
FROM dual
```

这段SQL语句是将字符串create table中每个单词的首字母都变为大写的格式。其查询的结果如下所示。

```
INITCAP('create table')
-----
Create Table
```

这里的查询结果为Create Table。可以看到，字符串Create Table是将SELECT语句中指定的字符串create table的每一个单词的首字母都变成了大写的结果。

**注意** MySQL数据库和Microsoft SQL Server数据库中不支持INITCAP函数。

### 10.1.5 截取字符串

在SQL语句中，如果希望对指定的字符串进行截取操作，在MySQL 5.0和Oracle数据库中可以使用SUBSTR函数。该函数的语法格式如下：

SUBSTR(string, start [,length])

该函数的功能是从指定的字符串string的start位置开始向后截取长度为length的子串。其中，参数string表示指定要截取的字符串；参数start表示指定要截取子串的起始位置；参数length用来指定要截取的子串的长度，[]里的参数length是可选的。参数start的值可以是正数，也可以是负数，如果是正数表示的是从指定字符串的起始位置（字符串的左边）开始算起，如果为负数，则表示从字符串的最后位置（字符串的右边）开始算起。下面是几个使用SUBSTR函数截取字符串的例子。

例10.6 截取指定字符串从第12个位置开始的子串。

```
SELECT SUBSTR('structured query language',12)
FROM dual
```

这段SQL语句是截取字符串structured query language从位置12开始，一直到字符串最后的子串。其查询结果如下所示。

```
SUBSTR('structured query language',12)
-----
query language
```

这里的查询结果为query language。可以看到，字符串query language是SELECT语句中指定的字符串structured query language从位置12开始截取一直到该字符串最后的子串。

例10.7 截取指定字符串从第12个位置开始，长度为5的子串。

```
SELECT SUBSTR('structured query language',12,5)
FROM dual
```

## 零基础学SQL

这段SQL语句是截取字符串structured query language中从位置12开始，长度为5的子串。其查询结果如下所示。

```
SUBSTR('structured query language',12,5)
-----
query
```

这里的查询结果为query。可以看到，字符串query是SELECT语句中指定的字符串structured query language从位置12开始截取长度为5的子串。

**例10.8** 截取指定字符串中的最后一个单词。

```
SELECT SUBSTR('structured query language',-8)
FROM dual
```

这段SQL语句是截取指定字符串中的最后一个单词。要想截取指定字符串中的最后一个单词，首先需要知道指定字符串中最后一个单词是什么，即需要知道最后一个单词字母的个数，然后再将取得的最后一个单词的字母个数前面加上一个负号，让其从字符串的后面开始截取指定长度的子串。其查询结果如下所示。

```
SUBSTR('structured query language',-8)
-----
language
```

这里的查询结果为language。可以看到，字符串language是SELECT语句中指定的字符串structured query language从最后的位置（字符串的最右侧）开始截取长度为8的子串。

**注意** Microsoft SQL Server数据库中使用的是SUBSTRING函数代替SUBSTR函数完成字符串截取的功能。其语法格式与SUBSTR函数的语法格式相同。

在MySQL数据库中SUBSTR函数除了可以使用上面讲到的语法格式截取字符串之外，还可以使用下面的语法格式截取指定的字符串。

```
SUBSTR(string FROM start[FOR length])
```

该函数的功能也是从指定的字符串string的start位置开始向后截取长度为length的子串。其中，参数string表示指定要截取的字符串；FROM表示指定字符串起始位置的关键字；参数start表示指定要截取子串的起始位置；FOR表示指定要截取的子串的长度关键字；参数length用来指定要截取的子串的长度。其中[]里面的FOR length是可选的。

例如，对于例10.6中截取指定字符串从第12个位置开始的子串的例子，使用上面讲的SUBSTRING语法格式，其SQL语句就可以使用下面的形式书写。

```
SELECT SUBSTR('structured query language' FROM 12)
FROM dual
```

这段SQL语句是截取字符串structured query language从位置12开始，一直到字符串最后的子串。这里使用FROM关键字代替例10.6中的逗号，其查询结果如下所示。

```
SUBSTR('structured query language' FROM 12)
-----
query language
```

其返回的结果与例10.6中显示的结果相同。读者可以应用SUBSTRING的这种语法格式使用MySQL数据库完成例10.7。

**注意** SUBSTR函数中，FOR关键字后指定的length的长度应为一个正数，如果length的值是一个负数，则返回的结果为空。

如果字符串中含有特殊字符，例如点号（.）、逗号（,）、冒号（:）等，在MySQL数据库中可以使用SUBSTRING\_INDEX函数。其语法格式如下：

```
SUBSTRING_INDEX(string,delim,count)
```

该函数的功能是截取指定字符串string从第count个位置开始出现的分隔符delim之后的子串。其中，参数string表示指定要截取的字符串，参数delim表示指定的分隔符，参数count表示要截取子串的位置。

参数count的值可以是正数，也可以是负数。如果参数count是正数，表示截取的子串是从该分隔符开始到指定字符串的起始位置（字符串的左边）的子串；如果参数count是负数，表示截取的是从该分隔符开始到指定字符串的最后位置（字符串的右边）的子串。

**例10.9** 截取指定的字符串（字符串中含有冒号，count值为正数）。

```
SELECT SUBSTRING_INDEX('structured:query:language',':',2)
FROM dual
```

这里给定的字符串中，每一个单词之间用冒号（:）分隔，其中count值指定为正数，表示截取的子串是从指定字符串的开始位置（字符串的左边）开始算起第二个位置出现的分隔符之前的子串。其查询的结果如下所示。

```
SUBSTRING_INDEX('structured:query:language',':',2)
-----
structured:query
```

从查询的结果可以看到，从指定字符串的开始位置（字符串的左边）开始算起第二个位置出现的分隔符是单词query和单词language之间的冒号。因此，最后查询的结果应该为从该冒号开始之前的子串，即structured:query。

**例10.10** 截取指定的字符串（字符串中含有冒号，count值为负数）。

```
SELECT SUBSTRING_INDEX('structured:query:language',':',-2)
FROM dual
```

这里给定的字符串中，每一个单词之间用冒号（:）分隔，其中count值指定为负数，表示截取的子串是从指定字符串的最后位置（字符串的右边）开始算起第二个位置出现的分隔符之后的子串。其查询的结果如下所示。

```
SUBSTRING_INDEX('structured:query:language',':',-2)
-----
query:language
```

从查询的结果可以看到，从指定字符串的最后位置（字符串的右边）开始算起第二个位置出现的分隔符是单词structured和单词query之间的冒号。因此，最后查询的结果应该为该冒号之后的子串，即query:language。



**注意** SUBSTRING\_INDEX函数中，表示分隔符的参数delim需要使用单引号将其引住。

### 10.1.6 从指定字符串的左侧读取子串

在SQL语句中，如果希望从指定字符串的左侧读取子串，可以使用LEFT函数。该函数的语法格式如下：

```
LEFT(string,length)
```

该函数的功能是取得指定字符串string中从左起length个长度的子串。其中，参数string表示要读取的指定字符串；参数length表示要取得的子串的长度。

**例10.11** 取得指定字符串最左面的10个子串。

```
SELECT LEFT('structured query language',10)
FROM dual
```

这段SQL要取得指定字符串中最左面的10个子串。其中，structured query language表示指定的字符串，数字10表示要取得的子串的长度为10。其查询结果如下所示。

```
LEFT('structured query language',10)
-----
structured
```

**注意** Oracle数据库不支持LEFT函数。

### 10.1.7 从指定字符串的右侧读取子串

在SQL语句中，如果希望从指定字符串的右侧读取子串，可以使用RIGHT函数。该函数的语法格式如下：

```
RIGHT(string,length)
```

该函数的功能是取得指定字符串string中从右起length个长度的子串。其中，参数string表示要读取的指定字符串；参数length表示要取得的子串的长度。

**例10.12** 取得指定字符串最右面的10个子串。

```
SELECT RIGHT('structured query language',8)
FROM dual
```

这段SQL要取得指定字符串中最右面的8个子串。其中，structured query language表示指定的字符串，数字8表示要取得的子串的长度为8。其查询结果如下所示。

```
RIGHT('structured query language',8)
-----
language
```

**注意** Oracle数据库不支持RIGHT函数。

### 10.1.8 去除字符串左侧空格或者字符

在SQL语句中，如果希望去除指定字符串中左侧的空格或者字符，可以使用LTRIM函数。该函数的语法格式如下：

```
LTRIM(string,[set])
```

该函数的功能是去除指定字符串中左侧含有的空格或者字符。其中，参数string表示要去除左侧空格的指定字符串；参数set表示要去除的字符。参数set是可选的。

**例10.13** 去除指定字符串左侧的空格。

```
SELECT LTRIM('    structured query language')
FROM dual
```

这段SQL语句是要去除指定字符串左侧的空格。其中，structured query language表示去除左侧空格的指定字符串，该字符串的前面包含有空格。其查询结果如下所示。

```
LTRIM('    structured query language')
-----
structured query language
```

从查询的结果可以看到，显示的结果中指定字符串structured query language中前面的几个空格已经去除了。

如果希望去除指定字符串中左侧的字符，就需要在LTRIM函数中再加上一个参数set，用来指定要去除的字符。

**例10.14** 去除指定字符串左侧的字符。

```
SELECT LTRIM('.structured query language','.')
FROM dual
```

这段SQL语句是要去除指定字符串左侧的字符。其中，.structured query language表示要去除左侧字符的指定字符串，该字符串前面有一个点号(.)。LTRIM函数中的第二个参数是用单引号引住的一个点号，表示的是要将指定字符串前面的点号去除。其查询结果如下所示。

```
LTRIM('.structured query language','.')
-----
structured query language
```

从查询的结果可以看到，显示的结果中指定字符串.structured query language中前面的点号已经被去除了。

MySQL数据库和Microsoft SQL Server数据库中使用LTRIM函数只能完成去除指定字符串左侧空格的功能。如果想要完成去除指定字符串左侧字符的功能，需要使用TRIM函数（TRIM函数的内容将在10.1.10节中介绍）。

### 10.1.9 去除字符串右侧空格或者字符

在SQL语句中，如果希望去除指定字符串中右侧的空格或者字符，可以使用RTRIM函数。该函数的语法格式如下：

`RTRIM(string,[set])`

该函数的功能是去除指定字符串中右侧含有的空格或者字符。其中，参数string表示要去除右侧空格的指定字符串；参数set表示要去除的字符。参数set是可选的。

例10.15 去除指定字符串右侧的空格。

```
SELECT RTRIM(' structured query language ')
FROM dual
```

这段SQL语句是要去除指定字符串右侧的空格。其中，structured query language表示去除右侧空格的指定字符串，该字符串的最后包含有空格。其查询结果如下所示。

```
RTRIM('structured query language ')
-----
structured query language
```

从查询的结果可以看到，显示的结果中指定字符串structured query language中后面的几个空格已经去除了。

如果希望去除指定字符串中右侧的字符，就需要在RTRIM函数中再加上一个参数set，用来指定要去除的字符。

例10.16 去除指定字符串右侧的字符。

```
SELECT RTRIM('structured query language.', '.')
FROM dual
```

这段SQL语句是要去除指定字符串右侧的字符。其中，structured query language.表示要去除右侧字符的指定字符串，该字符串前面有一个点号（.）。RTRIM函数中的第二个参数是用单引号引住的一个点号，表示的是要将指定字符串后面的点号去除。其查询结果如下所示。

```
RTRIM('structured query language.', '.')
-----
structured query language
```

从查询的结果可以看到，显示的结果中指定字符串structured query language.中后面的点号已经被去除了。

**注意** MySQL数据库和Microsoft SQL Server数据库中使用RTRIM函数只能完成去除指定字符串右侧空格的功能。在MySQL数据库如果想要完成去除指定字符串右侧字符的功能，需要使用TRIM函数。（TRIM函数的内容将在10.1.10小节中介绍）

### 10.1.10 去除字符串两侧空格或者字符

在SQL语句中，如果希望去除指定字符串中左侧和右侧两侧的空格或者字符，在Oracle数据库和MySQL数据库中可以使用TRIM函数。该函数的语法格式如下：

```
TRIM([BOTH | LEADING | TRAILING][set][FROM] string)
```

该函数的功能是去除指定字符串中左侧和右侧两侧的空格或者字符。其中，参数string表示要去除左侧和右侧两侧的空格或者字符的指定字符串；BOTH关键字表示要去除指定字符串左侧和右侧的空格，它是TRIM函数的默认选项；LEADING关键字表示去除指定字符串左侧的字符；TRAILING关键字

表示去除指定字符串右侧的字符；参数set是可选的，表示要去除的字符。关键字BOTH、TRAILING、TRAILING和FROM都是可选的。

**例10.17** 去除指定字符串左侧和右侧的空格。

```
SELECT TRIM('    structured query language    ')
FROM dual
```

这段SQL语句是要去除指定字符串左侧和右侧的空格。其中，structured query language表示去除左侧和右侧空格的指定字符串，该字符串的前面和后面都包含有空格。其查询结果如下所示。

```
TRIM ' structured query language '
-----
structured query language
```

从查询的结果可以看到，显示的结果中指定字符串structured query language中前面和后面的几个空格都被去除了。

如果希望去除指定字符串中左侧的字符，就需要在TRIM函数中加上一个参数set以及LEADING和FROM关键字，用来指定要去除字符串中左侧的字符。

**例10.18** 去除指定字符串左侧的字符。

```
SELECT TRIM(LEADING '.' FROM '...structured query language')
FROM dual
```

这段SQL语句是要去除指定字符串左侧的字符。其中，...structured query language表示要去除左侧字符的指定字符串，该字符串前面包含了三个点号(.)。TRIM函数中LEADING关键字表示要去除的是指定字符串左侧的字符；LEADING关键字后面跟的就是要去除的字符，这里指定的是点号(.)。这个点号需要使用单引号引住。其查询结果如下所示。

```
TRIM(LEADING '.' FROM '...structured query language')
-----
structured query language
```

从查询的结果可以看到，显示的结果中指定字符串...structured query language中左侧的三个点号已经被去除了。

如果希望去除指定字符串中右侧的字符，就需要在TRIM函数中加上一个参数set以及TRAILING和FROM关键字，用来指定要去除字符串中右侧的字符。

**例10.19** 去除指定字符串右侧的字符。

```
SELECT TRIM(TRAILING 'x' FROM 'structured query languagexxx')
FROM dual
```

这段SQL语句是要去除指定字符串右侧的字符。其中，structured query languagexxx表示要去除右侧字符的指定字符串，该字符串中的后面有三个字符x。TRIM函数中TRAILING 'x'关键字表示要去除的是指定字符串右侧的字符；TRAILING 'x'关键字后面跟的就是要去除的字符，这里指定的是字符“x”。这个字符“x”需要使用单引号引住。其查询结果如下所示。

```
TRIM(TRAILING 'x' FROM 'structured query languagexxx')
-----
structured query language
```



从查询的结果可以看到，显示的结果中指定字符串structured query language xxx中后面的的字符“x”已经被去除了。

**注意** Microsoft SQL Server数据库中，不支持TRIM函数。

### 10.1.11 左侧填充空格或者字符

在SQL语句中，如果希望在指定字符串的左侧填充空格或者字符，在Oracle数据库和MySQL数据库中可以使用LPAD函数。该函数的语法格式如下：

```
LPAD(string,length,padstring)
```

该函数的功能是在指定字符串的左侧填充空格或者字符。其中，参数string表示要填充空格或者字符的指定字符串；参数length表示字符串填充后返回的字符串的总长度；参数padstring表示要填充的字符。如果参数length的值为负数或者参数padstring的值为空，则LPAD函数的返回值为NULL。

**例10.20** 在指定字符串的左侧填充字符串SQL。

```
SELECT LPAD('select',9,'SQL')
FROM dual
```

这段SQL语句是要在指定字符串的左侧填充字符串SQL。其中，select表示要填充字符的指定字符串，数字9表示字符串填充后返回的字符串的总长度为9，SQL表示要填充的字符串。其查询结果如下所示。

```
LPAD('select',9,'SQL')
-----
SQLselect
```

从查询的结果可以看到，在指定的字符串select前面多出了字符串SQL，最终显示的字符串SQLselect的总长度为9。

对于例10.20的SQL语句查询的结果，其使用LPAD函数返回的字符串的总长度正好是指定字符串的长度与要填充的字符串的长度的和。那么，如果LPAD函数中指定的length值大于或者小于填充后返回的字符串的总长度，那最后的查询结果会有什么变化呢？下面仍以例10.20为例，修改length值，看一下查询的结果会有什么变化。

首先将length的值修改为10，其SQL语句如下：

```
SELECT LPAD('select',10,'SQL')
FROM dual
```

这段SQL语句是要在指定字符串的左侧填充字符串SQL。其中，select表示要填充字符的指定字符串，数字10表示字符串填充后返回的字符串的总长度为10，SQL表示要填充的字符串。其查询结果如下所示。

```
LPAD('select',10,'SQL')
-----
SQLSselect
```

可以看到，将LPAD 函数中的length值修改为10后，其查询出来的字符串变为SQLSselect，其字符串的总长度也是10。



将length的值修改为8，其SQL语句如下：

```
SELECT LPAD('select',8,'SQL')
FROM dual
```

这段SQL语句是要在指定字符串的左侧填充字符串SQL。其中，select表示要填充字符的指定字符串，数字8表示字符串填充后返回的字符串的总长度为8，SQL表示要填充的字符串。其查询结果如下所示。

```
LPAD('select',8,'SQL')
-----
SQselect
```

可以看到，将LPAD 函数中的length值修改为8后，其查询出来的字符串变为SQselect，其字符串的总长度也是8。

从LPAD函数的查询结果上看，LPAD 函数返回的字符串是从左面用指定字符串padding填充，直到该字符串的长度为length为止。

下面读者可以再考虑一个问题，上面的例子中，参数length的值都比要填充的指定字符串string的长度大，如果length的值小于指定字符串string的长度，那最后填充的效果会如何呢？下面将参数length的值修改为3，其SQL语句如下：

```
SELECT LPAD('select',3,'SQL')
FROM dual
```

这段SQL语句是要在指定字符串的左侧填充字符串SQL。其中，select表示要填充字符的指定字符串，数字3表示字符串填充后返回的字符串的总长度为3，SQL表示要填充的字符串。其查询结果如下所示。

```
LPAD('select',3,'SQL')
-----
sel
```

可以看到，将LPAD 函数中的length值修改为3后，其查询出来的字符串变为sel，其字符串的总长度为3。也就是说，如果length的值小于指定字符串string的长度，则使用LPAD 函数返回的就是指定字符串string从左侧算起的第length个字符。

Microsoft SQL Server数据库中不支持LPAD函数。如果希望在Microsoft SQL Server数据库中填充空格，可以使用space函数。space函数可以返回一个由N个长度的空格组成的字符串。例如space(10)表示由10个空格组成的字符串。

### 10.1.12 右侧填充空格或者字符

在SQL语句中，如果希望在指定字符串的右侧填充空格或者字符，在Oracle数据库和MySQL数据库中可以使用RPAD函数。该函数的语法格式如下：

```
RPAD(string,length,padstring)
```

该函数的功能是在指定字符串的右侧填充空格或者字符。其中，参数string表示要填充空格或者字符的指定字符串；参数length表示字符串填充后返回的字符串的总长度；参数padstring表示要填充的字符串。

符。如果参数length的值为负数或者参数padstring的值为空，则RPAD函数的返回值为NULL。

**例10.21** 在指定字符串的右侧填充字符串SQL。

```
SELECT RPAD('select',9,'SQL')
FROM dual
```

这段SQL语句是要在指定字符串的右侧填充字符串SQL。其中，select表示要填充字符的指定字符串，数字9表示字符串填充后返回的字符串的总长度为9，SQL表示要填充的字符串。其查询结果如下所示。

```
RPAD('select',9,'SQL')
-----
selectSQL
```

从查询的结果可以看到，在指定的字符串select的后面多出了一个字符串SQL，最终显示的字符串selectSQL的总长度为9。

对于例10.21的SQL语句查询的结果，其使用RPAD函数返回的字符串的总长度正好是指定字符串的长度与要填充的字符串的长度的和。那么，如果RPAD函数中指定的length值大于或者小于填充后返回的字符串的总长度，那最后的查询结果会有什么变化呢？下面仍以例10.21为例，修改length值，看一下查询的结果会有什么变化。

首先将length的值修改为10，其SQL语句如下：

```
SELECT RPAD('select',10,'SQL')
FROM dual
```

这段SQL语句是要在指定字符串的右侧填充字符串SQL。其中，select表示要填充字符的指定字符串，数字10表示字符串填充后返回的字符串的总长度为10，SQL表示要填充的字符串。其查询结果如下所示。

```
RPAD('select',10,'SQL')
-----
selectSQLS
```

可以看到，将RPAD函数中的length值修改为10后，其查询出来的字符串变为selectSQLS，其字符串的总长度也是10。

将length的值修改为8，其SQL语句如下：

```
SELECT RPAD('select',8,'SQL')
FROM dual
```

这段SQL语句是要在指定字符串的右侧填充字符串SQL。其中，select表示要填充字符的指定字符串，数字8表示字符串填充后返回的字符串的总长度为8，SQL表示要填充的字符串。其查询结果如下所示。

```
RPAD('select',8,'SQL')
-----
selectSQ
```

可以看到，将RPAD函数中的length值修改为8后，其查询出来的字符串变为selectSQ，其字符串的总长度也是8。

从RPAD函数的查询结果上看，RPAD函数返回的字符串是从右边用指定字符串padstring填充，直到该字符串的长度为length为止。

下面读者可以再考虑一个问题，上面的例子中，参数length的值都比要填充的指定字符串string的长度大，如果length的值小于指定字符串string的长度，那最后填充的效果会如何呢？下面将参数length的值修改为3，其SQL语句如下：

```
SELECT RPAD('select',3,'SQL')
FROM dual
```

这段SQL语句是要在指定字符串的右侧填充字符串SQL。其中，select表示要填充字符的指定字符串，数字3表示字符串填充后返回的字符串的总长度为3，SQL表示要填充的字符串。其查询结果如下所示。

```
RPAD('select',3,'SQL')
-----
sel
```

可以看到，将RPAD函数中的length值修改为3后，其查询出来的字符串变为sel，其字符串的总长度为3。也就是说，如果length的值小于指定字符串string的长度，则使用RPAD函数返回的就是指定字符串string从左侧算起的第length个字符。

**注意** Microsoft SQL Server数据库中不支持RPAD函数。

### 10.1.13 取得指定的子串在字符串中的位置

在SQL语句中，取得指定的子串在字符串中的位置，不同数据库有不同的实现方法。这里主要介绍在Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库中如何实现取得指定的子串在字符串中的位置的功能。

#### 1. Oracle数据库

在SQL语句中，如果希望取得指定的子串在字符串中的位置，可以使用INSTR函数。该函数的语法格式如下：

```
INSTR(string,substring, [,n [,m]])
```

该函数的功能是取得子串在指定的字符串中出现的位置。该函数返回的是一个整数。其中，参数string表示指定字符串；参数substring表示指定的子串；参数n表示指定字符串搜索的起始位置，如果n为正数，表示从字符串的左侧开始计算，如果n为负数，表示从字符串的右侧开始计算；参数m表示指定子串substring出现的次数。n和m的值都是可选的，它们的默认值都为1。下面来看几个使用INSTR函数的例子。

**例10.22** 查找字符u在指定字符串中第一次出现的位置。

```
SELECT INSTR('structured query language', 'u')
FROM dual
```

这段SQL语句是要查找字符u在指定字符串中第一次出现的位置。其中，structured query language表示指定的字符串，字符u表示要查找的子串，其查询结果如下所示。

```
INSTR('structured query language', 'u')
```

```
-----  
4
```

查询结果中返回的结果是4，表示子串u在字符串structured query language中第一次出现是在该字符串的第4个位置上。可以看到字符串structured query language中第一个单词“structured”中第一次出现字符u就在该单词的第4个位置上。

**例10.23** 从指定字符串第5个位置开始查找字符u第一次出现的位置。

```
SELECT INSTR('structured query language', 'u', 5)  
FROM dual
```

这段SQL语句是从指定字符串第5个位置开始查找字符u在指定字符串中第一次出现的位置。其中，structured query language表示指定的字符串，字符u表示要查找的子串，其查询结果如下所示。

```
INSTR('structured query language', 'u', 5)
```

```
-----  
7
```

查询结果中返回的结果是7，表示从字符串structured query language的第5个位置开始，子串u第一次出现是在该字符串的第7个位置上。可以看到字符串structured query language中第一个单词“structured”中，从第5个位置开始第一次出现字符u是在指定字符串的第7个位置上。

**例10.24** 从指定字符串第5个位置开始查找字符u第二次出现的位置。

```
SELECT INSTR('structured query language', 'u', 5, 2)  
FROM dual
```

这段SQL语句是从指定字符串第5个位置开始查找字符u在指定字符串中第二次出现的位置。其中，structured query language表示指定的字符串，字符u表示要查找的子串，其查询结果如下所示。

```
INSTR('structured query language', 'u', 5, 2)
```

```
-----  
13
```

查询结果中返回的结果是13，表示从字符串structured query language的第5个位置开始，子串u第二次出现是在该字符串的第13个位置上。可以看到第5个位置开始第二次出现字符u是在字符串structured query language中的第二个单词“query”上，该字符u在指定字符串的第13个位置上。

**例10.25** 从指定字符串的最后向前查找字符u第一次出现的位置。

```
SELECT INSTR('structured query language', 'u', -1)  
FROM dual
```

这段SQL语句是从指定字符串的最后向前查找字符u第一次出现的位置。其中，structured query language表示指定的字符串，字符u表示要查找的子串。其查询结果如下所示。

```
INSTR('structured query language', 'u', -1)
```

```
-----  
22
```

查询结果中返回的结果是22，表示从字符串structured query language的最后位置开始，子串u第一次出现是在该字符串的第22个位置上。可以看到从最后位置开始，第一次出现字符u是在字符串

structured query language中的第三个单词“language”上，该字符u在指定字符串的第22个位置上。

**注意** INSTR函数中，参数n表示指定字符串搜索的起始位置，n既可以是正数，也可以是负数。如果n为正数，表示从指定字符串的起始位置（字符串的左边）开始算起；如果n为负数，则表示从字符串的最后位置（字符串的右边）开始算起。

## 2. MySQL数据库

在MySQL数据库中，也有一个INSTR函数，用来取得指定的子串在字符串中的位置。但是该函数只有两个参数，其语法格式如下：

```
INSTR(string,substring)
```

该函数的功能是返回子串substring在指定字符串string中第一次出现的位置。其中，参数string表示指定字符串；参数substring表示指定的子串。如果指定的子串不在字符串string中，则返回0。

**例10.26** 查找字符v在指定字符串中第一次出现的位置。

```
SELECT INSTR('structured query language', 'v')
FROM dual
```

这段SQL语句是要查找字符v在指定字符串中第一次出现的位置。其中，structured query language表示指定的字符串，字符v表示要查找的子串，其查询结果如下所示。

```
INSTR('structured query language', 'v')
-----
0
```

查询结果中返回的结果是0，表示子串v在字符串structured query language中不存在。可以看到，在字符串structured query language中并不存在v这个字符。

在MySQL数据库中，还有一个与INSTR函数功能相同的函数POSITION函数。该函数的语法格式如下：

```
POSITION(substring IN string)
```

该函数的功能是返回子串substring在指定字符串string中第一次出现的位置。其中，参数substring表示指定的子串；参数string表示指定字符串，IN是该函数的一个关键字。如果指定的子串不在字符串string中，则返回0。

**例10.27** 查找子串SQL在指定字符串中第一次出现的位置（使用POSITION函数）。

```
SELECT POSITION('SQL' IN 'MySQL')
FROM dual
```

这段SQL语句是要查找子串SQL在指定字符串中第一次出现的位置。其中，子串SQL表示要查找的子串，MySQL表示指定的字符串。其查询结果如下所示。

```
POSITION('SQL' IN 'MySQL')
-----
3
```

查询结果中返回的结果是3，该函数返回的是子串SQL中的首字母S在指定的字符串MySQL第一次出现的位置。



在MySQL数据库中，另外一个和INSTR函数功能相似的函数是LOCATE函数。只不过在LOCATE函数中，参数string和参数substring颠倒了位置。其语法格式如下：

```
LOCATE(substring, string,[start])
```

该函数的功能也是返回子串substring在指定字符串string中第一次出现的位置。LOCATE函数中第一个参数要指定子串，第二个参数表示的是指定的字符串。其中，参数substring表示指定的子串；参数string表示指定字符串；参数start表示指定字符串搜索的起始位置，是一个正数。如果指定的子串不在字符串string中，则返回0。下面来看几个使用LOCATE函数的例子。

例10.28 查找子串SQL在指定字符串中第一次出现的位置（使用LOCATE函数）。

```
SELECT LOCATE('SQL','MySQL')
FROM dual
```

这段SQL语句是要查找子串SQL在指定字符串中第一次出现的位置。其中子串SQL表示要查找的子串，MySQL表示指定的字符串。其查询结果如下所示。

```
LOCATE('SQL',' MySQL ')
-----
3
```

查询结果中返回的结果是3，其查询结果与例10.27中查询的结果是相同的。

例10.29 从指定字符串第5个位置开始查找字符u第一次出现的位置（使用LOCATE函数）。

```
SELECT LOCATE('u','structured query language',5)
FROM dual
```

这段SQL语句是从指定字符串第5个位置开始查找字符u在指定字符串中第一次出现的位置。其中，structured query language表示指定的字符串，字符u表示要查找的子串，其查询结果如下所示。

```
LOCATE('structured query language','u',5)
-----
7
```

查询结果中返回的结果是7，其查询结果与例10.23中查询的结果是相同的。

### 3. Microsoft SQL Server数据库

在Microsoft SQL Server数据库中，并不支持INSTR函数，因此要想实现上面讲到的INSTR函数的功能，需要使用CHARINDEX函数。其语法格式如下：

```
CHARINDEX(substring, string,[start])
```

该函数的功能也是返回子串substring在指定字符串string中第一次出现的位置。CHARINDEX函数中第一个参数指定子串，第二个参数表示的是指定的字符串。其中，参数substring表示指定的子串；参数string表示指定字符串；参数start表示指定字符串搜索的起始位置，是一个正数。如果指定的子串不在字符串string中，则返回0。

例10.30 查找子串SQL在指定字符串中第一次出现的位置（使用CHARINDEX函数）。

```
SELECT CHARINDEX('SQL','Microsoft SQL Server')
```

这段SQL语句是要查找子串SQL在指定字符串中第一次出现的位置。其中子串SQL表示要查找的子

串，Microsoft SQL Server表示指定的字符串。其查询结果如下所示。

```
CHARINDEX('SQL','Microsoft SQL Server')
```

```
-----  
11
```

查询结果中返回的结果是11，该函数返回的是子串SQL中的首字母S在指定的字符串Microsoft SQL Server中第一次出现的位置。

在Microsoft SQL Server数据库中，还有一个函数可以实现取得指定的子串在字符串中的位置的功能，这就是PATINDEX函数。其语法格式如下：

```
PATINDEX('%substring% ', string)
```

该函数的功能也是返回子串substring在指定字符串string中第一次出现的位置。其中，参数substring表示指定的子串，该子串的前面和后面需要使用百分号（%）进行标记；参数string表示指定字符串。如果指定的子串不在字符串string中，则返回0。

**例10.31** 查找子串SQL在指定字符串中第一次出现的位置（使用PATINDEX函数）。

```
SELECT PATINDEX('%SQL%', 'Microsoft SQL Server')
```

这段SQL语句是要查找子串SQL在指定字符串中第一次出现的位置。其中子串SQL表示要查找的子串，在子串SQL的前面和后面需要有百分号（%）对其进行标记。Microsoft SQL Server表示指定的字符串。其查询结果如下所示。

```
PATINDEX('%SQL%', 'Microsoft SQL Server')
```

```
-----  
11
```

查询结果中返回的结果是11，该函数返回的是子串SQL中的首字母S在指定的字符串Microsoft SQL Server中第一次出现的位置。

Microsoft SQL Server数据库中，用CHARINDEX函数和PATINDEX函数都可以实现取得指定的子串在字符串中的位置的功能，但是PATINDEX函数支持使用通配符，可以在参数substring中使用通配符搜索字符串。

#### 注意

要实现取得指定的子串在字符串中的位置的功能，不同数据库有不同的实现方法。Oracle数据库中使用INSTR函数；MySQL数据库中虽然也可以使用INSTR函数，但是功能没有Oracle数据库中的INSTR函数强大，在MySQL数据库中还可以使用POSITION函数和LOCATE函数实现取得指定的子串在字符串中的位置的功能；在Microsoft SQL Server数据库中需要使用CHARINDEX函数或者PATINDEX函数实现取得指定的子串在字符串中的位置的功能。

### 10.1.14 颠倒指定字符串的顺序

在SQL语句中，如果希望颠倒指定字符串的顺序，可以使用REVERSE函数。该函数的语法格式如下：

```
REVERSE(string)
```

该函数的功能是颠倒指定字符串的顺序。其中，参数string表示指定要颠倒顺序的字符串。

### 例10.32 将指定字符串的顺序颠倒。

```
SELECT REVERSE('query')
FROM dual
```

这段SQL语句是颠倒指定字符串的顺序，将指定字符串按照从右向左的顺序输出。其中字符串query表示要颠倒顺序的字符串，其查询结果如下所示。

```
REVERSE('query')
-----
yreuq
```

查询结果中返回的结果是yreuq，显示的字符串按照与指定字符串相反的顺序，即从右向左的顺序显示输出。

### 10.1.15 替换指定的子串

在SQL语句中，如果希望将字符串中的某一子串用指定的子串替换掉，可以使用REPLACE函数。该函数的语法格式如下：

```
REPLACE(string, search_string [,replace_string])
```

该函数的功能是将字符串string中指定的子串search\_string都用子串replace\_string替换掉。其中，参数string表示指定要替换内容的字符串；参数search\_string表示被替换的子串；参数replace\_string表示要替换search\_string内容的字符串，参数replace\_string是可选的。如果省略replace\_string，则表示去除掉指定字符串string中search\_string子串所表示的内容。

### 例10.33 将指定子串替换掉。

```
SELECT REPLACE('structured query language','structured')
FROM dual
```

这段SQL语句是要将指定的子串structured去除。其中字符串structured query language表示指定的替换字符串，structured表示要被替换掉的子串，其查询结果如下所示。

```
REPLACE('structured query language','structured' )
-----
query language
```

查询结果中返回的结果是query language，从查询结果中可以看到指定字符串中structured query language的子串structured被去除了。

### 例10.34 将指定子串用字符串SQL替换。

```
SELECT REPLACE('structured query language','structured' ,'SQL')
FROM dual
```

这段SQL语句是要将指定的子串用字符串SQL替换。其中字符串structured query language表示指定的替换字符串，structured表示要被替换掉的子串，字符串SQL表示的是将子串structured的内容替换为字符串SQL。其查询结果如下所示。

```
REPLACE('structured query language','structured' ,'SQL')
-----
SQL query language
```

查询结果中返回的结果是SQL query language，从查询结果中可以看到指定字符串中structured query language的子串structured被字符串SQL替换掉了。

**注意** REPLACE函数中第三个参数replace\_string在Oracle数据库中是可选的，但是在MySQL数据库是需要指定的。如果在MySQL数据库和Microsoft SQL Server数据库中要实现例10.33中的功能，其SELECT语句可以这样写：SELECT REPLACE('structured query language','structured','')。第三个参数用一对单引号引起来，其最终的查询结果与例10.33中显示的结果是相同的。

在Microsoft SQL Server数据库中，如果希望将字符串中的某一子串用指定的子串替换掉，还可以使用STUFF函数。该函数的语法格式如下：

```
STUFF(string , start , length , replace_string )
```

该函数的功能是将字符串string中从指定的start位置开始，长度为length的子串用指定的字符串replace\_string替换掉。其中，参数string表示指定要替换内容的字符串；参数start表示要替换字符串的起始位置；参数length表示被替换子串的长度；参数replace\_string表示要替换内容的字符串。如果起始位置start的值或长度值length为负数，或者起始位置start的值比字符串string的长度值大，将返回NULL。

例如对于例10.34，如果使用STUFF函数，其SQL语句如下：

```
SELECT STUFF('structured query language',1,10,'SQL')
```

这段SQL语句是将字符串structured query language中从第1个字符，长度为10的子串用指定的字符串SQL替换掉。其查询结果如下所示。

```
STUFF('structured query language',1,10,'SQL')
-----
SQL query language
```

其查询结果与例10.34中查询结果相同。

### 10.1.16 字符替换

在SQL语句中，如果希望将字符串中指定的字符用另一个字符替换掉，在Oracle数据库中可以使用TRANSLATE函数。该函数的语法格式如下：

```
TRANSLATE(string, search_string, replace_string)
```

该函数的功能是将字符串string中指定的子串search\_string的每一个字符用字符串replace\_string中与其对应位置的字符替换掉。其中，参数string表示指定要替换内容的字符串；参数search\_string表示被替换的子串；参数replace\_string表示要替换search\_string内容的字符串。

**例10.35** 替换字符串中的指定字符。

```
SELECT TRANSLATE('structured query language','tr','TR')
FROM dual
```

这段SQL语句是要将指定的字符“t”用字符“T”替换掉，将指定的字符“r”用字符“R”替换掉。其中字符串structured query language表示指定的替换字符串。其查询结果如下所示。

```
TRANSLATE('structured query language','tr','TR')
-----
sTRucTuRed queRy language
```

查询结果中返回的结果是sTRucTuRed queRy language，从查询结果中可以看到指定字符串中structured query language所有的字符“t”都被字符“T”替换掉，所有的字符“r”都被字符“R”替换掉。

上面的例子中，被替换的字符与要替换search\_string内容的字符串中的字符是一一对应的，如果被替换的字符与要替换search\_string内容的字符串中的字符不是一一对应的，那查询的结果又会如何呢？来看下面这个例子。

#### 例10.36 替换字符串中的指定字符。

```
SELECT TRANSLATE('structured query language','tr','T')
FROM dual
```

这段SQL语句中替换的子串有两个字符，分别是字符“t”和字符“r”。而要替换子串内容的字符串中却只有一个字符，字符串structured query language表示指定的替换字符串。来看一下这个SQL语句的查询结果。

```
TRANSLATE('structured query language','tr','T')
```

```
-----
sTucTued quey language
```

查询结果中返回的结果是sTucTued quey language，从查询结果中可以看到指定字符串中structured query language所有的字符“t”都被字符“T”替换掉，字符“r”由于没有与之对应的字符，所以用空替换掉。

**注意** MySQL数据库和Microsoft SQL Server数据库不支持TRANSLATE函数。

### 10.1.17 拼接字符串

在SQL语句中，如果希望将两个字符串拼接在一起，可以使用CONCAT函数。该函数在Oracle数据库中的语法格式如下：

```
CONCAT(string1, string2)
```

该函数的功能是将两个字符串拼接在一起并返回拼接后的新的字符串。其中，参数string1表示要连接的第一个字符串；参数string2表示要连接的第二个字符串。

#### 例10.37 在学生信息表中将学生姓名和年龄连接起来，中间用冒号分开。

```
SELECT CONCAT(CONCAT(stuName, ':'),age)
FROM t_student
```

这段SQL语句是将学生信息表中学生姓名和年龄连接起来，并在学生姓名和年龄之间用冒号将其区分开。由于在Oracle数据库中，CONCAT函数一次只能连接两个字符串，所以在这里需要使用两个CONCAT函数。其中内层的CONCAT函数首先将表示学生姓名的字段stuName和冒号连接起来，然后外层的CONCAT函数再将学生的姓名和冒号连接起来的字符串作为一个新的字符串和表示学生年龄的字段age连接，组成一个新的字符串。其查询结果如下所示。

```
CONCAT(CONCAT(stuName, ':'),age)
```

```
-----
赵亮:23
```



```
郑茹:21
王海:23
张明:22
王昌鹤:24
王玉梅:22
李玉峰:24
李凤:24
```

从显示的结果中可以看到，显示的新的字符串中学生的姓名和年龄连接在一起，在学生的姓名和年龄之间用冒号将它们区分开来。

Oracle数据库中虽然可以使用CONCAT函数对字符串进行连接操作，但是从上面的例子可以看到该函数一次只能连接两个字符串，如果希望将多个字符串连接在一起，就需要使用多个CONCAT函数一起使用，这样连接很不方便。

在MySQL数据库中，也提供了一个可以拼接字符串的CONCAT函数，该函数可以接受多个字符串参数。其语法规则如下：

```
CONCAT(string1, string2, string3,...)
```

该函数的功能是可以将多个字符串拼接在一起并返回拼接后的新的字符串。其中，参数string1表示要连接的第一个字符串；参数string2表示要连接的第二个字符串；参数string3表示要连接的第三个字符串，以此类推，直到将参数中指定字符串全部拼接完成。

下面还以例10.37为例，如果在MySQL数据库中使用CONCAT函数，其SQL语句就可以使用以下的方法完成。

```
SELECT CONCAT(stuName, ': ', age)
FROM t_student
```

这段SQL语句中，只使用了一个CONCAT函数，将学生信息表中表示学生姓名的字段stuName、作为分隔符的冒号以及表示学生年龄的字段age连接到一起。其查询结果如下所示。

```
+-----+-----+
| CONCAT(stuName, ': ', age) |
+-----+-----+
| 赵亮:23                    |
| 郑茹:21                    |
| 王海:23                    |
| 张明:22                    |
| 王昌鹤:24                  |
| 王玉梅:22                  |
| 李玉峰:24                  |
| 李凤:24                    |
+-----+-----+
```

其显示的结果与例10.37中显示的结果相同。

在MySQL数据库中使用CONCAT函数，如果CONCAT函数的参数中有一个值为NULL，则CONCAT函数最终返回的结果也是NULL。例如下面的SQL语句。

```
SELECT CONCAT(stuName, NULL, age)
FROM t_student
```

这里将连接学生姓名和年龄中间的冒号换成NULL值，其他的内容不变，来看一下显示的结果会发

## 零基础学SQL

生什么样的变化。其查询结果如下所示。

+-----+-----+		
CONCAT(stuName, NULL age)		
+-----+-----+		
	NULL	
	NULL	
	NULL	
	NULL	
	NULL	
	NULL	
	NULL	
	NULL	

可以看到，其显示的结果值都变为了NULL。

**注意** Microsoft SQL Server数据库中不支持CONCAT函数。如果在Microsoft SQL Server数据库中想要连接两个字符串，可以直接使用加号（+）进行连接。

### 10.1.18 取得字符的ASCII码

在SQL语句中，如果希望取得某一个字符的ASCII码，可以使用ASCII函数。该函数的语法格式如下：

```
ASCII(string)
```

该函数的功能是取得字符串中最左面的一个字符的ASCII码。其中，参数string表示指定的字符串。如果字符串string是一个空字符串，则返回0；如果字符串string是NULL，则返回NULL值。

**例10.38** 取得字符Z的ASCII码。

```
SELECT ASCII('Z')
FROM dual
```

这段SQL语句获取的是单个字符Z的ASCII码。字符Z需要用单引号将其引住。其查询的结果如下所示。

```
ASCII('Z')
-----
90
```

其查询结果为90，就是字符Z在ASCII码表中对应的值。

如果ASCII函数中不是一个字符，而是一个字符串，那么使用ASCII函数会返回该字符串中最左面的一个字符的ASCII码。例如下面这个例子。

**例10.39** 取得字符串ABC的ASCII码。

```
SELECT ASCII('ABC')
FROM dual
```

这段SQL语句获取的是字符串ABC的ASCII码。字符串ABC需要用单引号将其引住。其查询的结果如下所示。

```
ASCII('ABC')
```

65

其查询结果为65，就是该字符串ABC第一个大写字母A在ASCII码表中对应的值。

ASCII函数中，如果指定的字符串都是由数字组成的，那么该字符串可以不使用单引号。例如，对于字符串987来说，使用SELECT语句，SELECT ASCII(987)和SELECT ASCII('987')其查询结果是一样的。如果字符串不都是由数字组成的，也就是说字符串中含有字符的话，那么该字符串就必须使用单引号引住，否则执行该SQL语句就会出现错误。

### 10.1.19 将ASCII码转换为相应的字符

在SQL语句中，如果希望将ASCII码转换为相应的字符，在Oracle数据库中可以使用CHR函数。该函数的语法格式如下：

CHR(N)

该函数的功能是返回指定的ASCII码对应的字符。ASCII码值需要是一个在0~255之间的数。其中，参数N表示指定的ASCII码。如果N值为NULL，则返回的结果为NULL。

例10.40 查询65对应的字符。

```
SELECT CHR(65)
FROM dual
```

这段SQL语句要取得ASCII码值为65对应的字符。其查询的结果如下所示。

CHR(65)

A

其查询结果为字符A，就是ASCII码表中值65对应的字符。

在Microsoft SQL Server数据库和MySQL数据库中可以使用CHAR函数将ASCII码转换为相应的字符。该函数的语法格式如下：

CHAR(N)

该函数的功能是返回指定的ASCII码对应的字符。ASCII码值需要是一个在0~255之间的数。其中，参数N表示指定的ASCII码。如果N值为NULL，则返回的结果为NULL。该函数的功能和使用方法和CHR函数的相同，这里就不再举例了，读者可以在Microsoft SQL Server数据库或者是MySQL数据库中使用CHAR函数查询某一个ASCII码对应的值。

### 10.1.20 匹配发音

在SQL语句中，如果希望区分两个字符串的发音是否相同，可以使用SOUNDEX函数。该函数的语法格式如下：

SOUNDEX(string)

该函数的功能是区分两个字符串的发音是否相同。该函数返回的是该字符串发音的语音表示。如果两个字符串的发音相同，则它们应该返回的是同一个值。其中，参数string表示指定的字符串。

例10.41 给出两个单词hart和heart，查询这两个单词的发音是否相同。



为了对这两个单词的发音进行比较，需要分别使用SOUNDEX查询它们的返回值。首先查询单词hart返回的语言表示，其SQL语句如下：

```
SELECT SOUNDEX('hart')
FROM dual
```

这段SQL语句是查询单词hart返回的语言表示。其查询结果如下所示。

```
SOUNDEX('hart')
-----
H630
```

再查询单词heart返回的语言表示，其SQL语句如下：

```
SELECT SOUNDEX('heart')
FROM dual
```

这段SQL语句是查询单词heart返回的语言表示。其查询结果如下所示。

```
SOUNDEX('heart')
-----
H630
```

这两个单词通过使用SOUNDEX函数，其返回的结果都是H630。从两个SOUNDEX函数的返回结果可以看到，单词hart和单词heart拥有同样的发音。从这个例子可以看出，使用SOUNDEX函数对比较那些单词拼写不同，但是读音相同的单词是很有用的。

### 10.1.21 将字符串重复指定次数

在SQL语句中，如果希望将字符串重复输出显示，在MySQL数据库中使用REPEAT函数。该函数的语法格式如下：

```
REPEAT(string ,count)
```

该函数的功能是指定字符串string重复count值，并返回重复count次后的字符串。其中，参数string表示指定要重复显示的字符串；参数count表示重复的次数。如果参数count <= 0，返回一个空字符串；如果参数string或者是count的值是NULL，则返回NULL。

例10.42 将指定字符串重复2次后显示（REPEAT函数）。

```
SELECT REPEAT('MySQL',2)
FROM dual
```

这段SQL语句是将指定的字符串MySQL重复2次，并将重复2次后的字符串显示输出。其查询结果如下所示。

```
+-----+-----+
| REPEAT('MySQL',2) |
+-----+-----+
| MySQL MySQL      |
```

从显示的查询结果可以看到，显示的字符串MySQL MySQL是由SELECT语句中指定的字符串MySQL重复2次后组成的一个新的字符串。

在Microsoft SQL Server数据库中，如果希望将字符串重复输出显示，需要使用REPLICATE函数。

其语法格式如下：

```
REPLICATE(string ,count)
```

该函数的功能与MySQL数据库中的REPEAT函数功能相同。其中，参数string表示指定要重复显示的字符串；参数count表示重复的次数。如果参数count <= 0，返回一个空字符串；如果参数string或者是count的值是NULL，则返回NULL。

例10.43 将指定字符串重复2次后显示（REPLICATE函数）。

```
SELECT REPLICATE('SQL Server',2)
```

这段SQL语句是将指定的字符串SQL Server重复2次，并将重复2次后的字符串显示输出。其查询结果如下所示。

```
REPLICATE('SQL Server',2)
-----
SQL Server SQL Server
```

从显示的查询结果可以看到，显示的字符串SQL Server SQL Server是由SELECT语句中指定的字符串SQL Server重复2次后组成的一个新的字符串。

**注意** Oracle数据库中不支持REPEAT函数和REPLICATE函数。

## 10.2 数字函数

数字函数中，输入的参数和函数的返回值都是一个数字类型的值。数字函数有很多，不同的数据库中完成相同功能的字符函数也不完全相同。本节将介绍一些在实际中经常用到的数字函数。这里以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，对于一些数字函数在这几种数据库中的一些不同的使用方法都做了比较详细的介绍。

### 10.2.1 求绝对值

在SQL语句中，如果希望计算某一个数字对应的绝对值，可以使用ABS函数。该函数的语法格式如下：

```
ABS(n)
```

该函数的功能是计算某一个数字对应的绝对值，该函数返回的值是一个非负数。其中，参数n表示指定的数字。如果参数n的值是NULL，则该函数的返回值也为NULL。

例10.44 计算指定数字的绝对值。

```
SELECT ABS(-3)
FROM dual
```

这段SQL语句是计算指定数字-3的绝对值。其查询的结果如下所示。

```
ABS(-3)
-----
3
```



### 10.2.2 求平方

在SQL语句中，如果希望计算某一个数字的平方值，在Microsoft SQL Server数据库中提供了一个计算数字平方值的SQUARE函数。该函数的语法格式如下：

```
SQUARE(n)
```

该函数的功能是计算某一个数字的平方值，该函数返回的值是一个非负数。其中，参数n表示指定的数字。如果参数n的值是NULL，则该函数的返回值也为NULL。

例10.45 计算指定数字的平方值。

```
SELECT SQUARE(-3)
```

这段SQL语句是计算指定数字-3的平方值。其查询的结果如下所示。

```
SQUARE(-3)
```

```
-----  
9
```

在Oracle数据库和MySQL数据库中，如果希望计算某一个数字的平方值，可以使用POWER函数。该函数在10.2.5小节中做详细介绍。

**注意** 如果希望计算某一个数字的平方值，在Microsoft SQL Server数据库中可以使用SQUARE函数，在Oracle数据库和MySQL数据库中可以使用POWER函数。

### 10.2.3 求平方根

在SQL语句中，如果希望计算某一个数字对应的平方根，可以使用SQRT函数。该函数的语法格式如下：

```
SQRT(n)
```

该函数的功能是计算某一个数字对应的平方根，该函数返回的值是一个非负数。其中，参数n表示指定的数字，其值必须大于等于0。如果参数n的值为负数或者是NULL，则该函数的返回值也为NULL。

例10.46 计算指定数字的平方根。

```
SELECT SQRT(9)  
FROM dual
```

这段SQL语句是计算指定数字9的平方根。其查询的结果如下所示。

```
SQRT(9)
```

```
-----  
3
```

### 10.2.4 求对数

首先了解一下对数的定义，如果 $a^b=N$ （ $a$ 大于0，且 $a$ 不等于1），那么数 $b$ 叫做以 $a$ 为底 $N$ 的对数，记作 $\log_a N=b$ ，其中 $a$ 叫做对数的底数， $N$ 叫做真数。其中以 $e$ 为底的对数叫做自然对数（ $e=2.7182818284590451$ ）， $N$ 的自然对数记作 $\ln N(N>0)$ 。

由于求对数函数在不同的数据库中有不同的表示方法，所以这里分别以求自然对数和以10为底的对数为例，介绍在Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库等不同的数据库中对数的使用方法。

### 1. 求自然对数

在Microsoft SQL Server数据库和MySQL数据库中，如果希望计算以e为底的自然对数的值，可以使用LOG函数。该函数的语法规则如下：

LOG(n)

该函数的功能是计算指定数字n的自然对数的值。其中，参数n表示指定的数字，其值必须大于等于0。如果参数n的值为负数或者是NULL，则该函数的返回值也为NULL。

例10.47 计算指定数字的自然对数。

```
SELECT LOG(2.7182818284590451) AS value1, LOG(9) AS value2
FROM dual
```

这段SQL语句是计算e和数字9的自然对数的值。根据自然对数的定义，如果自然对数中，真数N的值为e（e=2.7182818284590451）的话，则该自然对数的值应该为1。其查询的结果如下所示。

value1	value2
1	2.1972245773362

从查询结果可以看到，在LOG函数中，如果参数n的值为2.7182818284590451（即无理数e的值），则其返回的结果为1；如果参数n为其他的大于等于0的值，则会根据对数的运算法则，计算出一个值。

在Oracle数据库中，如果希望计算以e为底的自然对数的值可以使用LN函数。该函数的语法格式如下：

LN(n)

该函数的功能是计算指定数字n的自然对数的值。其中，参数n表示指定的数字，其值必须大于等于0。如果参数n的值为负数或者是NULL，则该函数的返回值也为NULL。

该函数的功能和使用方法与前面讲的LOG函数的使用方法相同，读者可以在Oracle数据库中使用该函数计算一下例10.47中给定数字的自然对数的值。这里就不再举例了。

**注意** 如果要计算以e为底的自然对数的值，在Microsoft SQL Server数据库和MySQL数据库中可以使用LOG函数；在Oracle数据库中，可以使用LN函数。

### 2. 求以10为底的对数

在Microsoft SQL Server数据库和MySQL数据库中，如果希望计算以10为底的自然对数的值，可以使用LOG10函数。该函数的语法规则如下：

LOG10(n)

该函数的功能是计算指定数字n的以10为底的对数值。其中，参数n表示指定的数字，其值必须大于等于0。如果参数n的值为负数或者是NULL，则该函数的返回值也为NULL。

例10.48 计算指定数字以10为底的对数值。

## 零基础学SQL

```
SELECT LOG10(10) AS value1, LOG10(9) AS value2
FROM dual
```

这段SQL语句是计算数字10和数字9的以10为底的对数值。根据对数的定义，对于以10为底的对数，如果真数N的值为10，则该对数的值应该为1。其查询的结果如下所示。

value1	value2
1	0.95424250943932

从查询结果可以看到，在LOG10函数中，如果参数n的值为10，则其返回的结果为1；如果参数n为其他的大于等于0的值，则会根据对数的运算法则，计算出一个值。

在Oracle数据库中，可以使用LOG函数计算以a为底n的对数的值。该函数的语法格式如下：

```
LOG(a,n)
```

该函数的功能是计算以a为底n的对数的值。其中，参数a表示对数的底数，其值可以是除0和1以外的正整数；参数n表示对数的真数，其值可以是任何正整数。如果参数a或者参数n的值为负数或者是NULL，则该函数的返回值也为NULL。

如果想用该函数计算以10为底的对数值，将参数a的值定义为10即可。读者可以在Oracle数据库中，使用该函数计算一下例10.48中给定数字的自然对数的值。这里就不再举例了。

**注意** 如果要计算以10为底的自然对数的值，在Microsoft SQL Server数据库和MySQL数据库中可以使用LOG10函数；在Oracle数据库中，可以将LOG函数中第一个参数的值设定为10。

### 10.2.5 求幂

首先了解一下幂运算，a的b次幂，可以写成 $a^b$ 。其中a叫做底数，b叫做指数。幂运算的逆运算是对数运算。

#### 1. 以e为底的幂运算

在Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库中，都提供了以e为底的幂运算的函数。如果希望计算e的n次幂的值，可以使用EXP函数。该函数的语法格式如下：

```
EXP(n)
```

该函数的功能是计算e的n次幂的值。其中，参数n表示指定的幂指数。如果参数n的值是NULL，则该函数的返回值也为NULL。

例10.49 计算e的1次幂和e的3次幂。

```
SELECT EXP(1),EXP(3)
FROM dual
```

这段SQL语句是计算e的1次幂和e的3次幂值。其查询的结果如下所示。

EXP(1)	EXP(3)
2.7182818284590451	20.085536923188

从查询结果可以看到，在EXP函数中，如果参数n的值为1，则其返回的结果为无理数e的值

2.7182818284590451；如果参数n为其他的值，则会根据幂运算的运算法则，计算出一个小数值。

## 2. 以其他任意数为底的幂运算

在SQL语句中，如果希望以其他任意数为底的幂运算，可以使用POWER函数。该函数的语法格式如下：

```
POWER(a,b)
```

该函数的功能是计算数字a的b次幂的值。其中，参数a表示底数，参数b表示指定的幂指数。如果参数b的值为2，则计算的就是数字a的平方值；如果参数b的值为3，则计算的就是数字a的立方值。如果参数a或者b的值是NULL，则该函数的返回值也为NULL。

例10.50 计算-2的3次幂和-2的-3次幂。

```
SELECT POWER(-2,3), POWER(-2,-3)
FROM dual
```

这段SQL语句是计算-2的3次幂和-2的-3次幂的值。其查询的结果如下所示。

```
POWER(-2,3)    POWER(-2,-3)
-----
-8              -0.125
```

从查询结果可以看到，在POWER函数中，参数a和参数b的值可以为负数，如果参数b为负数，则会根据幂运算的运算法则，计算的应该是 $1/a^b$ 的值。

**注意** 在MySQL数据库中，还可以使用POW函数进行求幂运算，其功能和使用方法与POWER函数的功能和使用方法相同。

## 10.2.6 对指定值进行四舍五入操作

在SQL语句中，如果希望对指定值进行四舍五入操作，可以使用ROUND函数。该函数的语法格式如下：

```
ROUND(n [,m])
```

该函数的功能是返回指定数字n的四舍五入后的结果。其中，参数n表示指定要进行四舍五入操作的数字；参数m是可选的，它可以是正整数，也可以是负整数。如果m是正整数，表示四舍五入到小数点右边第m位；如果m是负整数，表示四舍五入到小数点左边的第m位，当m的绝对值大于数字n的整数部分时，返回的结果为0；如果省略m或者m的值为0，表示四舍五入到数字n的整数位。如果参数n或者m的值为负数或者是NULL，则该函数的返回值也为NULL。

例10.51 对指定数字进行四舍五入操作。

```
SELECT ROUND(3.5), ROUND(2345.3654879,3)
FROM dual
```

这段SQL语句是对指定数字3.5和2345.3654879进行四舍五入的操作。其查询的结果如下所示。

```
ROUND(3.5)    ROUND(2345.3654879,3)
-----
4              2345.365
```

## 零基础学SQL

在这个例子中，第一个是对数字3.5进行四舍五入的计算，使用了省略参数m的ROUND函数，第二个使用了带有参数m的ROUND函数计算指定数字中四舍五入到小数点后第3位的值。这里，参数m的值为正数，如果参数m的值为负数，那么对指定数字进行四舍五入操作的结果会如何呢？来看下面这个例子。

**例10.52** 对指定数字进行四舍五入操作。

```
SELECT ROUND(2345.3654879,-3), ROUND(2345.3654879,-4)
FROM dual
```

这段SQL语句是对指定数字2345.3654879进行四舍五入的操作，其中ROUND函数中参数m的值为负数。其查询的结果如下所示。

```
ROUND(2345.3654879,-3)  ROUND(2345.3654879,-4)
-----
2000                      0
```

从查询结果可以看到，对于数字2345.3654879来说，当ROUND函数中参数m的值为-3时，其显示的结果为2000；当ROUND函数中参数m的值为-4时，其显示的结果却变为0。读者考虑一下，当m的值为负数时，为什么会产生这样的结果。

**注意** ROUND函数中，如果m是负数，表示四舍五入到小数点左边的第m位；当m的绝对值大于数字n的整数部分时，返回的结果为0。

如果把数字2345.3654879改为2545.3654879，那么当m的值为-3时，ROUND函数中查询的结果会有什么变化？如果把数字2345.3654879改为5345.3654879，那么当m的值为-4时，查询的结果又会如何呢？

### 10.2.7 求两数相除的余数

在SQL语句中，如果希望取得两数相除的余数，在Oracle数据库和MySQL数据库中可以使用MOD函数。该函数的语法格式如下：

```
MOD(n,m)
```

该函数的功能是计算两数相除的余数。其中，参数n表示被除数；参数m表示除数。如果参数m的值为0，则返回参数n对应的值；如果参数n或者m的值是NULL，则该函数的返回值也为NULL。

**例10.53** 计算两数相除的余数。

```
SELECT MOD(10,6)
FROM dual
```

这段SQL语句是计算数字10和6的余数。其查询的结果如下所示。

```
MOD(10,6)
-----
4
```

MOD函数功能与%运算符的功能是相同的。在Microsoft SQL Server数据库中，如果希望计算两数相除的余数，可以直接使用%运算符。

**注意** 如果希望计算两数相除的余数，在Oracle数据库和MySQL数据库中可以使用MOD函数，也可以使用%运算符，在Microsoft SQL Server数据库中没有与之对应的函数，可以使用%运算符。



### 10.2.8 取得大于等于指定数的最小整数

在SQL语句中，如果希望取得大于等于指定数的最小整数，在Oracle数据库和MySQL数据库中可以使用CEIL函数。该函数的语法格式如下：

```
CEIL(n)
```

该函数的功能是返回大于等于指定数字n的最小整数。其中，参数n表示指定的数字。如果参数n的值是NULL，则该函数的返回值也为NULL。

**例10.54** 取得大于等于指定数的最小整数。

```
SELECT CEIL(10.3)
FROM dual
```

这段SQL语句是取得大于等于指定数字10.3的最小整数。其查询的结果如下所示。

```
CEIL(10.3)
-----
11
```

在Microsoft SQL Server数据库和MySQL数据库中，如果希望取得大于等于指定数的最小整数，可以使用CEILING函数，其功能和使用方法与CEIL函数相同。

**注意** 如果希望取得大于等于指定数的最小整数，在Microsoft SQL Server数据库和MySQL数据库中可以使用CEILING函数，在Oracle数据库和MySQL数据库中可以使用CEIL函数。

### 10.2.9 取得小于等于指定数的最大整数

在SQL语句中，如果希望取得小于等于指定数的最小整数，可以使用FLOOR函数。该函数的语法格式如下：

```
FLOOR(n)
```

该函数的功能是返回小于等于指定数字n的最大整数。其中，参数n表示指定的数字。如果参数n的值是NULL，则该函数的返回值也为NULL。

**例10.55** 取得小于等于指定数的最小整数。

```
SELECT FLOOR(10.3)
FROM dual
```

这段SQL语句是取得小于等于指定数字10.3的最大整数。其查询的结果如下所示。

```
FLOOR(10.3)
-----
10
```

### 10.2.10 求正弦与余弦值

在SQL语句中，如果希望取得指定数字的正弦值，可以使用SIN函数。该函数的语法格式如下：

```
SIN(n)
```

该函数的功能是返回指定数字n的正弦值。其中，参数n表示的是弧度数。如果参数n的值是NULL，

## 零基础学SQL

则该函数的返回值也为NULL。

**例10.56** 计算指定数字的正弦值。

```
SELECT SIN(PI()/6)
FROM dual
```

这段SQL语句是计算弧度为 $\pi/6$ 的正弦值。其中，函数PI()表示圆周率 $\pi$ 。其查询的结果如下所示。

```
SIN(PI()/6)
```

```
-----
0.5
```

从查询结果可以看到，SIN(PI()/6)的返回值是0.5。SIN(PI()/6)也就是计算SIN( $\pi/6$ )的值，根据弧度和角度之间的关系可以知道，弧度 $\pi/6$ 对应的角度值为30度。因此，SIN(PI()/6)也就是求SIN30°的值，根据正弦值求值的计算规则，SIN30°的值是1/2，即0.5。

如果希望取得指定数字的余弦值，可以使用COS函数。该函数的语法格式如下：

```
COS(n)
```

该函数的功能是返回指定数字n的余弦值。其中，参数n表示的是弧度数。如果参数n的值是NULL，则该函数的返回值也为NULL。

**例10.57** 计算指定数字的余弦值。

```
SELECT COS(PI()/3)
FROM dual
```

这段SQL语句是计算弧度为 $\pi/3$ 的余弦值。其中，函数PI()表示圆周率 $\pi$ 。其查询的结果如下所示。

```
COS(PI()/3)
```

```
-----
0.5
```

从查询结果可以看到，COS(PI()/3)的返回值是0.5。COS(PI()/3)也就是计算COS( $\pi/3$ )的值，根据弧度和角度之间的关系可以知道，弧度 $\pi/3$ 对应的角度值为60度。因此，COS(PI()/3)也就是求COS 60°的值，根据余弦值求值的计算规则，COS 60°的值是1/2，即0.5。

### 10.2.11 求正切值与余切值

在SQL语句中，如果希望取得指定数字的正切值，可以使用TAN函数。该函数的语法格式如下：

```
TAN(n)
```

该函数的功能是返回指定数字n的正切值。其中，参数n表示的是弧度数。如果参数n的值是NULL，则该函数的返回值也为NULL。

**例10.58** 计算指定数字的正切值。

```
SELECT TAN(PI()/4)
FROM dual
```

这段SQL语句是计算弧度为 $\pi/4$ 的正切值。其中，函数PI()表示圆周率 $\pi$ 。其查询的结果如下所示。

```
TAN(PI()/4)
```

```
-----
1
```

从查询结果可以看到， $\text{TAN}(\text{PI}()/4)$ 的返回值是1。 $\text{TAN}(\text{PI}()/4)$ 也就是计算 $\text{TAN}(\pi/4)$ 的值，根据弧度和角度之间的关系可以知道，弧度 $\pi/4$ 对应的角度值为45度。因此， $\text{TAN}(\text{PI}()/4)$ 也就是求 $\text{TAN}45^\circ$ 的值，根据正切值求值的计算规则， $\text{TAN}45^\circ$ 的值是1。

如果希望取得指定数字的余切值，可以使用COT函数。该函数的语法格式如下：

**COT(n)**

该函数的功能是返回指定数字n的余切值。其中，参数n表示的是弧度数。如果参数n的值是NULL，则该函数的返回值也为NULL。

**例10.59 计算指定数字的余切值。**

```
SELECT COT(PI()/4)
FROM dual
```

这段SQL语句是计算弧度为 $\pi/4$ 的余切值。其中，函数PI()表示圆周率 $\pi$ 。其查询的结果如下所示。

```
COT(PI()/4)
-----
1
```

从查询结果可以看到， $\text{COT}(\text{PI}()/4)$ 的返回值是1。 $\text{COT}(\text{PI}()/4)$ 也就是计算 $\text{COT}(\pi/4)$ 的值，根据弧度和角度之间的关系可以知道，弧度 $\pi/4$ 对应的角度值为45度。因此， $\text{COT}(\text{PI}()/4)$ 也就是求 $\text{COT}45^\circ$ 的值，根据余切值求值的计算规则， $\text{COT}45^\circ$ 的值是1。

### 10.2.12 求反正弦和反余弦值

在SQL语句中，如果希望取得指定数字的反正弦值，可以使用SIN函数。该函数的语法格式如下：

**ASIN(n)**

该函数的功能是返回指定数字n的反正弦值，是一个弧度制。其中，参数n的范围在-1到1之间（包括-1和1）。如果参数n的值是NULL，则该函数的返回值也为NULL。

**例10.60 计算指定数字的反正弦值。**

```
SELECT ASIN(1)
FROM dual
```

这段SQL语句是计算1的反正弦值。其查询的结果如下所示。

```
ASIN(1)
-----
1.5707963267949
```

从查询结果可以看到，ASIN(1)的返回值是1.5707963267949。根据反正弦值求值的计算规则，对于反正弦函数写成 $y=\arcsin x$ 。当x的值为1时，y的值为 $\pi/2$ 。这里返回的值1.5707963267949就是 $\pi/2$ 的值。（其中 $\pi=3.1415926535897931\dots$ ）

如果希望取得指定数字的反余弦值，可以使用ACOS函数。该函数的语法格式如下：

**ACOS(n)**

该函数的功能是返回指定数字n的反余弦值，是一个弧度制。其中，参数n的范围在-1到1之间（包括-1和1）。如果参数n的值是NULL，则该函数的返回值也为NULL。

## 零基础学SQL

### 例10.61 计算指定数字的反余弦值。

```
SELECT ACOS(1)
FROM dual
```

这段SQL语句是计算1的反余弦值。其查询的结果如下所示。

```
ACOS(1)
-----
0
```

从查询结果可以看到，ACOS(1)的返回值是0。根据反正弦值求值的计算规则，对于反余弦函数写 $y = \arccos x$ 。当x的值为1时，y的值为0。这里ACOS(1)函数返回的值就是0。

### 10.2.13 求反正切值

在SQL语句中，如果希望取得指定数字的反正切值，可以使用ATAN函数和ATAN2函数。这两个函数的语法格式如下：

```
ATAN(n)
ATAN2(n,m)
```

其中，ATAN函数的功能是返回指定数字n的反正切值，是一个弧度制。其中，参数n可以是任何数字。ATAN2函数的功能是返回数字n除以数字m的正切值，是一个弧度制。其中，参数m的值不能为0。如果函数中参数n或者参数m的值是NULL，则该函数的返回值也为NULL。

### 例10.62 计算指定数字的反正切值。

```
SELECT ATAN(1),ATAN2(10,4)
FROM dual
```

这段SQL语句中，ATAN函数接收一个参数，计算数字1的反正切值。ATAN2函数接收2个参数。计算10/4的反正切值。其查询的结果如下所示。

```
ATAN(1)                ATAN2(10,4)
-----
0.78539816339745      1.1902899496825
```

从查询结果可以看到，数字1的反正切值和10/4的反正切值都是小数。根据反正切函数的性质，反正切函数是单调递增的，由于 $1 < 10/4$ ，因此，查询结果中ATAN(1)的值要小于ATAN(10,4)的值。

**注意** Microsoft SQL Server数据库不支持ATAN2函数。

### 10.2.14 弧度与角度的互换

在实际使用三角函数时，有些时候需要进行角度和弧度之间的互换。在Microsoft SQL Server数据库和MySQL数据库中提供了可以对角度和弧度进行互换的函数。本节就来介绍如何实现对角度和弧度的互换操作。

#### 1. 将弧度转换为角度

在Microsoft SQL Server数据库和MySQL数据库中，如果希望将弧度转换为角度，可以使用DEGREES函数。该函数的语法规则如下：

#### DEGREES(n)

该函数的功能是将指定的弧度转换为角度。其中，参数n表示指定要转换的弧度数。如果参数n的值是NULL，则该函数的返回值也为NULL。

**例10.63** 将指定的弧度数转换为角度。

```
SELECT DEGREES(PI()/3)
FROM dual
```

这段SQL语句是计算弧度PI()/3对应的角度值。其查询的结果如下所示。

```
DEGREES(PI()/3)
-----
60
```

从查询结果可以看到，DEGREES(PI()/3)的返回值是60。DEGREES(PI()/3)也就是计算 $\pi/3$ 对应的角度值，根据弧度和角度之间的转换关系可以知道，弧度 $\pi/3$ 对应的角度值为60度。因此，DEGREES(PI()/3)的返回值也是60。

#### 2. 将角度转换为弧度

在Microsoft SQL Server数据库和MySQL数据库中，如果希望将角度转换为弧度，可以使用RADIANS函数。该函数的语法规则如下：

#### RADIANS(n)

该函数的功能是将指定的角度转换为弧度。其中，参数n表示指定要转换的角度值。如果参数n的值是NULL，则该函数的返回值也为NULL。

**例10.64** 将指定的角度值转换为弧度。

```
SELECT RADIANS(60)
FROM dual
```

这段SQL语句是计算60度角对应的弧度数。其查询的结果如下所示。

```
RADIANS(60)
-----
1.0471975511966
```

从查询结果可以看到，RADIANS(60)的返回值是1.0471975511966。RADIANS(60)也就是计算60度角对应的弧度数，根据弧度和角度之间的转换关系可以知道，60度角对应的弧度数为 $\pi/3$ 。因此，RADIANS(60)的返回值也就是 $\pi/3$ 对应的值。

**注意** Oracle数据库中不支持DEGREES函数和RADIANS函数。

#### 10.2.15 取得指定值的符号标志

在SQL语句中，如希望对指定数字的正负号进行检测，可以使用SIGN函数。该函数的语法规则如下：

#### SIGN(n)

该函数的功能是对指定数字进行检测，取得该值的符号标志。如果给定的数字n小于0，则函数会



## 零基础学SQL

返回-1；如果给定的数字n大于0，则函数会返回1；如果给定的数字n等于0，则函数会返回0；如果参数n的值是NULL，则该函数的返回值也为NULL。

**例10.65** 取得指定值的符号标志。

```
SELECT sign(-6),sign(0),sign(6)
FROM dual
```

这段SQL语句是取得指定数字-6、数字0和数字6对应的符号标志。其查询的结果如下所示。

sign(-6)	sign(0)	sign(6)
-1	0	1

从查询结果可以看到，sign(-6)中，参数-6是负数，其返回的值是-1；sign(6)中，参数6是正数，其返回的值是1；sign(0)返回的值是0。

### 10.2.16 对指定值进行截取操作

在SQL语句中，如果希望指定值进行截取操作，在Oracle中可以使用TRUNC函数。该函数的语法格式如下：

```
TRUNC(n[,m])
```

该函数的功能是截取数字。其中，参数n表示要截取的数字；参数m是可选的，其值可以是正整数，也可以是负整数。如果是正整数，表示将指定的数字n截取到小数点右侧第m位；如果是负整数，表示将指定的数字n截取到小数点左侧第m位；如果m被省略或者m的值为0，表示将指定数字n的小数部分截取掉。如果参数n或者m的值是NULL，则该函数的返回值也为NULL。

**例10.66** 对指定数字进行截取。

```
SELECT TRUNC(10.6789),TRUNC(10.6789,2)
FROM dual
```

这段SQL语句是对指定的数字10.6789进行截取操作。其查询的结果如下所示。

TRUNC(10.6789)	TRUNC(10.6789,2)
10	10.67

在MySQL数据库中，如果希望指定值进行截取操作，可以直接使用TRUNCATE函数。该函数的功能和使用方法与TRUNC函数的相同，这里就不再举例了。读者可以在MySQL数据库中使用TRUNCATE函数实现例10.66中的截取操作。

#### 注意

如果希望对指定值进行截取操作，在Oracle数据库中使用TRUNC函数，在MySQL数据库中使用TRUNCATE函数。Microsoft SQL Server数据库中没有与之对应的函数。

## 10.3 日期时间函数

在实际应用中，经常要对日期时间进行处理，而日期的计算并不是使用加号运算符或者减号运算符进行简单的加减操作。在SQL语句中，对日期时间的处理需要使用与日期时间操作有关的函数来完

成。不同的数据库都提供了对日期时间进行处理的函数。这一节就以Oracle数据库、MySQL数据库和Microsoft SQL Server数据库为例，讲解常用的日期时间函数在这几种数据库中的使用方法。

### 10.3.1 取得当前系统的日期和时间

在数据库中提供了获取当前系统的日期和时间的函数。不同的数据库获取当前系统的日期和时间的函数也不完全相同。这里以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，介绍在这3种数据库中如何使用日期和时间函数取得当前系统的日期和时间。

#### 1. Oracle数据库

在Oracle数据库中，可以使用SYSDATE函数取得当前数据库系统的日期和时间。该函数可以返回当前系统的日期和时间。

**例10.67** 返回系统当前的日期和时间。

```
SELECT SYSDATE
FROM dual
```

这段SQL语句中，是使用SYSDATE函数来返回当前数据库系统的日期和时间，其查询结果如下所示。

```
SYSDATE
-----
2009-8-15 10:17:17
```

从查询结果可以看到，使用SYSDATE返回的日期和时间就是数据库系统当前的日期和时间。其默认格式是YYYY-MM-DD HH:MM:SS。即使用SYSDATE函数查询时，当前数据库系统的时间是2009年8月15日10点17分17秒。

在Oracle数据库中，可以使用CURRENT\_DATE函数获取数据库系统当前的日期，其显示的格式为YYYY-MM-DD。还可以使用CURRENT\_TIMESTAMP函数获取数据库系统当前的日期和时间。该函数返回的日期和时间跟设定的时区有关。

#### 2. MySQL数据库

在MySQL数据库中，可以使用NOW函数和SYSDATE函数取得当前数据库系统的日期和时间。该函数可以返回当前系统的日期和时间。

**例10.68** 返回系统当前的日期和时间。

```
SELECT NOW(),SYSDATE()
FROM dual
```

这段SQL语句中，是使用NOW函数来返回当前数据库系统的日期和时间，其查询结果如下所示。

```
+-----+-----+
|      NOW()      |      SYSDATE()      |
+-----+-----+
| 2009-8-15 10:30:35 | 2009-8-15 10:30:35 |
+-----+-----+
```

从查询结果可以看到，使用NOW函数和SYSDATE函数返回的日期和时间就是数据库系统当前的日期和时间。即使用NOW函数和SYSDATE函数查询时，当前数据库系统的时间是2009年8月15日10点30

分35秒。其默认的显示格式为YYYY-MM-DD HH:MM:SS。

使用NOW函数和SYSDATE函数取得数据库当前的系统时间，其显示结果是相同的。其显示的格式与Oracle数据库中显示的格式相同。

在MySQL数据库中，还可以使用CURRENT\_DATE函数和CURRENT\_TIME函数获取当前数据库系统中的日期和时间信息，其中CURRENT\_DATE函数可以获取到当前数据库系统中的日期信息，CURRENT\_TIME函数可以获取当前系统中的时间信息。

**例10.69** 分别显示当前数据库系统的日期和时间。

```
SELECT CURRENT_DATE, CURRENT_TIME
FROM dual
```

这段SQL语句中，使用CURRENT\_DATE函数来返回当前数据库系统的日期，使用CURRENT\_TIME函数返回当前系统的时间。其查询结果如下所示。

CURRENT_DATE	CURRENT_TIME
2009-8-15	10:53:50

从查询结果可以看到，CURRENT\_DATE函数返回的日期就是数据库系统当前的日期。其显示的日期格式为YYYY-MM-DD。CURRENT\_TIME函数返回当前系统的时间，其显示的格式为HH:MM:SS。

在MySQL数据库中，也可以使用CURRENT\_TIMESTAMP 获取数据库系统当前的日期和时间。该函数返回的日期和时间跟设定的时区有关。

### 3. Microsoft SQL Server数据库

在Microsoft SQL Server数据库中，可以使用GETDATE函数取得当前数据库系统的日期和时间。该函数可以返回当前系统的日期和时间。

**例10.70** 返回系统当前的日期和时间。

```
SELECT GETDATE()
```

这段SQL语句中，使用GETDATE函数来返回当前数据库系统的日期和时间，其查询结果如下所示。

```
GETDATE()
-----
2009-8-15 10:58:20
```

从查询结果可以看到，使用GETDATE函数返回的日期和时间就是数据库系统当前的日期和时间。即使用NOW函数查询时，当前数据库系统的时间是2009年8月15日10点58分20秒。其默认的显示格式为YYYY-MM-DD HH:MM:SS。显示的格式与Oracle数据库中显示的格式相同。

### 10.3.2 对日期值进行加减运算

有些时候，开发人员或者用户希望对取得的指定日期值进行相应的加减运算。不同的数据库对日期值进行加减运算的方法也不完全相同。这里以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，介绍在这3种数据库中如何使用日期和时间函数对日期值进行加减运算。

## 1. Oracle数据库

在Oracle数据库中，可以直接使用加号（+）运算符或者减号（-）运算符对指定日期值进行加减运算。

**例10.71** 取得数据库当前的系统时间后1天的日期时间。

```
SELECT SYSDATE, SYSDATE+1
FROM dual
```

这段SQL语句中，是使用加号（+）运算符将当前数据库系统中取得的日期值中的天数加上1天，其查询结果如下所示。

SYSDATE	SYSDATE+1
2009-8-15 11:17:17	2009-8-16 11:17:17

从查询结果可以看到，使用SYSDATE+1后，其日期值变为了2009-8-16 11:17:17。即比SYSDATE函数获取的当前数据库系统的时间多了1天，也就是第二天的日期时间。

如果想对指定日期值中的月份进行加减操作，可以使用Oracle数据库提供的ADD\_MONTH函数。该函数的语法规则如下：

```
ADD_MONTH(date,n)
```

该函数的功能是返回指定日期中，之前月份或者之后月份对应的日期时间信息。其中，参数date表示指定的日期时间；参数n是一个整数，可以是正整数，也可以是一个负整数。如果参数n是一个正整数，则该函数返回指定日期之后月份对应的日期时间信息；如果参数n是一个负整数，则该函数返回指定日期之前月份对应的日期时间信息。

**例10.72** 取得数据库当前的系统时间之前月份和之后月份对应的日期时间信息。

```
SELECT ADD_MONTH(SYSDATE,-1), ADD_MONTH(SYSDATE,1)
FROM dual
```

这段SQL语句中，使用ADD\_MONTH函数对数据库当前的系统时间的月份进行加减运算。其中ADD\_MONTH函数中第二个参数-1表示取得数据库当前的系统时间之前月份对应的日期时间信息，ADD\_MONTH函数中第二个参数1表示取得数据库当前的系统时间之后月份对应的日期时间信息。其查询结果如下所示。

ADD_MONTH(SYSDATE,-1)	ADD_MONTH(SYSDATE,1)
2009-7-15 11:44:23	2009-9-15 11:44:23

从查询结果可以看到，使用ADD\_MONTH(SYSDATE,-1)函数后，其显示的时间是数据库当前的系统时间之前月份对应的日期时间信息，使用ADD\_MONTH(SYSDATE,1)函数其显示的时间是数据库当前的系统时间之后月份对应的日期时间信息。

**注意** Oracle数据库中日期值和日期值之间不能进行相加运算，但日期值和日期值之间可以进行减法运算。日期值和数字之间可以进行加法运算。

## 2. MySQL数据库

在MySQL数据库中，可以使用DATE\_ADD()对指定的日期时间进行增加一个特定的日期段或者时

零基础学SQL

间段。函数的语法格式如下：

```
DATE_ADD(date,INTERVAL expression type)
```

该函数的功能是对指定的日期时间增加一个特定的日期段或时间段。其中，参数date 表示指定的日期时间，它可以是一个 DATETIME 类型的值，也可以是一个DATE类型的值；参数expression是一个表达式，用来表示对指定日期date增加的时间间隔。INTERVAL 和type是两个关键字，type关键字用来指定表达式被解释的方式。

type关键字可以是DAY、MONTH、YEAR等。type关键字对应的值及其在DATE\_ADD函数中所表示的意义如表10.1所示。

表10.1 type关键字对应的值及其在DATE\_ADD函数中所表示的意义

type关键字对应的值	在DATE_ADD函数中所表示的意义
YEAR	对指定的日期时间值增加特定的年数
MONTH	对指定的日期时间值增加特定的月数
QUARTER	对指定的日期时间值增加特定的季度（1个季度为3个月）
WEEK	对指定的日期时间值增加特定的星期数
DAY	对指定的日期时间值增加特定的日子
HOURL	对指定的日期时间值增加特定的小时数
MINUTE	对指定的日期时间值增加特定的分钟数
SECOND	对指定的日期时间值增加特定的秒数
MICOSECOND	对指定的日期时间值增加特定的微秒数
YEAR_MONTH	对指定的日期时间值增加特定的年数和月数（'yy-mm'）
DAY_HOUR	对指定的日期时间值增加特定的小时数（'hh'）
DAY_MINUTE	对指定的日期时间值增加特定的小时和分钟数（'hh:mm'）
DAY_SECOND	对指定的日期时间值增加特定的小时、分钟和秒数（'hh:mm:ss'）

增加的日期时间的值是由DATE\_ADD函数中参数expression表达式对应的值来确定的。例如在DATE\_ADD函数中如果第二个参数写成INTERVAL 1 YEAR的形式，则表示对指定的日期值增加1年；在DATE\_ADD函数中如果第二个参数写成INTERVAL '01:10'DAY\_MINUTE的形式，则表示对指定的时间值增加1小时10分钟。

**注意** 关键词INTERVAL和type关键字对应的值均不区分大小写。

例10.73 取得数据库当前的系统时间后1天的日期时间。

```
SELECT NOW(),DATE_ADD(NOW(), INTERVAL 1 DAY) AS NEXTDAY
FROM dual
```

这段SQL语句中，使用NOW函数取得当前系统的日期时间值，使用DATE\_ADD函数将取得的当前系统的日期的天数加1。其中，DATE\_ADD函数中的第一个参数表示的是系统当前的日期值，第二个参数中，INTERVAL是关键字，DAY表示要增加日期值中的日子，1表示日期值中的日子要增加1天。其查询结果如下所示。

```
+-----+-----+-----+-----+
```



NOW()	NEXTDAY
2009-8-15 15:28:50	2009-8-16 15:28:50

从查询结果可以看到，当前的系统日期时间是2009年8月15日15点28分50秒，使用DATE\_ADD函数后，其显示的日期时间为2009年8月16日15点28分50秒，即比NOW函数获取的当前数据库系统的日期多了1天。

使用DATE\_ADD函数也可以为指定的日期和时间值增加一个特定的时间，下面来看一个这方面的例子。

**例10.74** 将取得的数据库当前的系统日期时间增加一个指定的时间。

```
SELECT NOW(),DATE_ADD(NOW(), INTERVAL '01:10:08' DAY_SECOND) AS NEWDAY
FROM dual
```

这段SQL语句中，使用NOW函数取得当前系统的日期时间值，使用DATE\_ADD函数为取得的当前系统的日期时间增加1小时10分08秒。其中，DATE\_ADD函数中的第一个参数表示的是系统当前的日期值，第二个参数中，INTERVAL是关键字，DAY\_SECOND表示要增加的时间值，字符串01:10:08表示增加的具体时间。其查询结果如下所示。

NOW()	NEWDAY
2009-8-15 16:16:16	2009-8-15 17:26:24

从查询结果可以看到，当前的系统日期时间是2009年8月15日16点16分16秒，使用DATE\_ADD函数后，其显示的日期时间为2009年8月15日17点26分24秒，即比NOW函数获取的当前数据库系统的时间多出了1小时10分08秒。

**注意** 对于DATE\_ADD函数中的参数expression，如果expression表达式中的值是一个正值，则表示对指定的日期和时间增加一个特定的日期段或时间段；如果expression表达式中的值是一个负值，则表示对指定的日期和时间减少一个特定的日期段或时间段。

在MySQL数据库中，如果希望对指定的日期和时间减去一个日期段或者是时间段，可以使用DATE\_SUB()对指定。函数的语法格式如下：

```
DATE_SUB (date,INTERVAL expression type)
```

该函数的功能是对指定的日期和时间减去一个特定的日期段或者是时间段。其中，参数date表示指定的日期时间，它可以是一个DATETIME类型的值，也可以是一个DATE类型的值；参数expression是一个表达式，用来表示对指定日期date减少的时间间隔。INTERVAL和type是两个关键字，type关键字用来指定表达式被解释的方式。type关键字对应的值及其在DATE\_ADD函数中所表示的意义与表10.1所示的内容相同，这里就不再重述了。

**例10.75** 取得数据库当前的系统时间之前3个月的日期时间。

```
SELECT NOW(),DATE_SUB(NOW(), INTERVAL 3 MONTH ) AS NEWDAY
FROM dual
```

## 零基础学SQL

这段SQL语句中，使用NOW函数取得当前系统的日期时间值，使用DATE\_SUB函数取得当前系统的日期时间之前3个月的日期时间。其中，DATE\_SUB函数中的第一个参数表示的是系统当前的日期时间值，第二个参数中，INTERVAL是关键字，MONTH表示减少指定的日期中对应的月份，3表示显示的日期是指定的日期时间之前3个月的日期。其查询结果如下所示。

NOW()	NEWDAY
2009-8-15 16:08:38	2009-5-15 16:08:38

从查询结果可以看到，当前的系统日期时间是2009年8月15日16点08分38秒，使用DATE\_SUB函数后，其显示的日期时间为2009年5月15日16点08分38秒，即显示的是数据库当前的系统时间之前3个月的日期时间。

使用DATE\_SUB函数也可以为指定的日期和时间值减少一个特定的时间，其使用方法与DATE\_ADD函数的使用方法相同，读者可以参考例10.74中的例子，使用DATE\_SUB函数完成为指定的日期和时间值减少一个特定的时间的操作。

### 注意

在MySQL数据库中，对指定的日期和时间增加或者减少一个特定的日期段或时间段的函数还有ADDDATE函数、ADDTIME函数、SUBDATE函数和SUBTIME函数。但是在实际使用中，一般都是用DATE\_ADD函数代替ADDDATE函数和ADDTIME函数，DATE\_SUB函数代替SUBDATE函数和SUBTIME函数。

### 3. Microsoft SQL Server数据库

在Microsoft SQL Server数据库中，可以使用DATEADD()对指定的日期时间进行增加或者减少一个特定的日期段或者时间段。函数的语法格式如下：

```
DATEADD (datepart,number,date)
```

该函数的功能是对指定的日期时间增加一个特定的日期段或时间段。其中，参数datepart表示指定的日期时间中的某一部分；参数number表示要增加的值，该值应该是一个整数，如果参数number的值是一个小数，则其值只保留整数部分，舍弃小数点以后的部分。如果是正整数，则表示为指定的日期时间增加一个特定的日期段或时间段；如果是负整数，则表示为指定的日期时间减少一个特定的日期段或时间段；参数date表示返回日期或是时间格式的字符串表达式。

参数datepart的值可以是DAY、MONTH、YEAR等。参数datepart的值及其在DATEADD函数中所表示的意义如表10.2所示。

**例10.76** 将当前系统日期向后推迟两天。

```
SELECT DATEADD(DAY,2,GETDATE())
```

这段SQL语句中，使用DATEADD函数表示要在当前系统的日期值上加上两天并将取得的新的日期值返回。DATEADD函数中，第一个参数DAY表示指定日期中的日子，第二个参数2表示要在指定日期中日子值上加上2，第三个参数GETDATE函数指定当前系统时间。其查询结果如下所示。

```
DATEADD(day,2,GETDATE())
```

2009-8-15 16:58:20

表10.2 参数datepart的值及其在DATEADD函数中所表示的意义

datepart的值	缩写	在DATEADD函数中所表示的意义
YEAR	yy,yyyy	表示指定的年数
MONTH	m,mm	表示指定的月数
DAYOFYEAR	dy,y	表示指定年中的日子
QUARTER	qq,q	表示指定的季度（1个季度为3个月）
WEEK	wk,ww	表示指定一年中的星期数
DAY	d,d	表示指定的日子
HOURL	hh,h	表示指定的小时数
MINUTE	mi,n	表示指定的分钟数
SECOND	ss,s	表示指定的秒数
MICROSECOND	mcs	表示指定的微秒数

另外，在Microsoft SQL Server数据库中，还可以使用DATEDIFF函数返回两个指定日期的差值。该函数的语法规则如下：

DATEDIFF (datepart, date1,date2)

该函数的功能是计算第二个日期值date2与第一个日期值date1的差值。其中，参数datepart表示指定的日期时间中的某一部分；date1指定第一个日期值；date2指定第二个日期值。

例10.77 计算两个日期差值。

SELECT DATEDIFF(MONTH,'20090710','20090815' )

这段SQL语句中，使用DATEDIFF函数计算两个指定日期值的差值。在DATEDIFF函数中，第一个参数指定日期的月份，表示以月为单位来计算日期的时间间隔；第二个参数和第三个参数分别指定了两个日期值。其查询结果如下所示。

DATEDIFF(MONTH,'20090710','20090815' )

1

10.3.3 取得日期之后指定工作日对应的日期

在Oracle数据库中，可以使用NEXT\_DAY函数取得日期之后指定工作日对应的日期。该函数的语法规则如下：

NEXT\_DAY(date,n)

该函数的功能是返回指定的日期date之后第一个工作日（由参数n指定）对应的日期。其中，参数date表示指定的日期时间；参数n用来指定一周之内的某一天作为工作日。参数n的值要与数据库中设定的日期语言相匹配。这里假定将数据库中的日期语言设置为中文。

Oracle中，将日期语言改写成中文简体，可以使用以下的命令：

alter SESSION set NLS\_DATE\_LANGUAGE='SIMPLIFIED CHINESE'

## 零基础学SQL

其中字符串SIMPLIFIED CHINESE表示以简体中文的形式显示日期结果。简体中文中月份是以中文汉字的形式来显示。下面来看一下，在语言为简体中文下使用NEXT\_DAY函数取得日期之后指定工作日对应的日期的例子。

**例10.78** 取得当前系统日期时间之后下周三对应的日期。

```
SELECT SYSTEM, NEXT_DAY (SYSTEM, '星期三')
FROM dual
```

这段SQL语句中，使用SYSTEM函数取得当前系统的日期时间值，使用NEXT\_DAY函数取得当前系统日期时间之后下周三对应的日期。其查询结果如下所示。

SYSTEM	NEXT_DAY (SYSTEM, '星期三')
15-8月-09	19-8月-09

从查询的结果中可以看到，结果中的日期值是以中文简体的形式显示的。使用NEXT\_DAY (SYSTEM, '星期三')得到的显示结果是19-8月-09，即当前系统日期时间之后下周三对应的日期为2009年8月19日。

**注意** NEXT\_DAY函数中参数n也可以是数字。其中n为1表示周日，n为2表示周一，n为3表示周二，以此类推。

### 10.3.4 取得日期值中的指定内容

在实际应用中，有时开发人员或者用户对指定日期值中的某一部分感兴趣，例如日期值中的年份或者月份等。此时就需要获取日期值中的指定内容。不同的数据库获取日期值中的指定内容方法也不完全相同。这里以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，介绍在这3种数据库中如何使用日期和时间函数取得指定日期值中的指定内容。

#### 1. Oracle数据库

在Oracle数据库中，可以使用EXTRACT函数，获取指定日期值的某一个部分对应的值。该函数的语法规则如下：

```
EXTRACT(date FROM datetime)
```

该函数的功能是从指定的日期时间中，取得特定的日期或者时间值，如年份、月份等。其中，参数date表示指定要获取的日期时间数据，该数据可以是YEAR、MONTH、DAY等；FROM是关键字；关键字FROM后面的参数datetime用来指定日期时间值。如果datetime是日期类型的值，可以抽取日期中的年、月、日的信息，如果想获取时、分、秒等信息，需要在关键字FROM后面加上一个TIMESTAMP关键字，并且在TIMESTAMP关键字后面需要有表示时间的字符串。

**例10.79** 取得当前数据库系统时间中的年份。

```
SELECT EXTRACT (YEAR FROM SYSTEM)
FROM dual
```

这段SQL语句是取得当前数据库系统时间中的年份，其中，EXTRACT函数中的YEAR表示要获取当前日期时间中的年份，SYSTEM表示当前数据库的系统时间，该函数表示从当前数据库系统时间中

取得年份信息。其查询结果如下所示。

```
EXTRACT (YEAR FROM SYSTEM)
-----
2009
```

从查询的结果中可以看到，EXTRACT函数返回的值是2009，该值也就是当前数据库系统时间中的年份。

在Oracle数据库中，还可以使用TO\_CHAR函数，获取指定日期值的某一个部分对应的值。该函数的语法规则如下：

```
TO_CHAR(date,[,fmt])
```

该函数中第一个参数date表示指定的日期时间值，参数fmt可以决定取得指定日期值的某一个部分对应的值。下面通过一个例子来看一下TO\_CHAR获取指定日期值的某一个部分对应的值的用法。

例10.80 取得指定日期中的年份。

```
SELECT SYSTEM, TO_CHAR(SYSTEM, 'YYYY')
FROM dual
```

这段SQL语句中，是取得当前数据库系统时间中对应的年份。使用SYSTEM函数取得当前系统的日期时间值，使用TO\_CHAR函数取得当前系统日期时间中的年份值。其查询结果如下所示。

```
SYSTEM          TO_CHAR(SYSTEM, 'YYYY')
-----
2009-8-15 17:33:37      2009
```

从查询的结果中可以看到，结果中的值经过TO\_CHAR(SYSTEM,'YYYY')函数的转换，显示的2009就是当前数据库系统时间中对应的年份。

**注意** TO\_CHAR函数是一个字符转换函数，它既可以显示日期值，也可以对字符串进行转换，有关该函数的详细用法可以参看10.4.1节。

2. MySQL数据库

在MySQL数据库中，提供了一些函数用来获取指定日期值的某一个部分对应的值。这些函数的名称及其各自的含义如表10.3所示。

表10.3 取得指定日期值的某一个部分对应的值函数的名称及其各自的含义

函 数	含 义
DAYOFYEAR(date)	返回一年当中的某一天（1~366）
DAYOFMONTH(date)	返回一月当中的某一天（1~31）
DAYOFWEEK(date)	返回星期数（1~7）。其中1表示星期日，7表示星期六
WEEKOFYEAR(date)	返回日期对应的星期数（1~53）
DAYNAME(date)	返回实际中的某一天。如Monday、Tuesday等
MONTHNAME(date)	返回月份的名字。如January、February等
YEAR(date)	返回日起值中的年份（1000~9999）
MONTH(date)	返回月份值（1~12）





(续)

函 数	含 义
WEEKDAY (date)	返回星期值 (0~6) 其中0代表星期一，6代表星期日
WEEK(date)	返回指定时间在一年当中是第几个星期
WEEK(date,first)	返回指定时间在一年当中是第几个星期
HOUR(time)	返回时间值中的小时数 (0~23)
MINUTE(time)	返回时间值中的分钟数 (0~59)
SECOND(time)	返回时间值中的秒数 (0~59)

❑ 在WEEK(date) 函数中，参数date表示指定的日期值。如果返回值为1，则1表示一年当中的第一个星期，其中星期日是每一个星期的第一天。

❑ 在WEEK(date,first) 函数中，参数date表示指定的日期值；参数first用来指定每一个星期的第一天是从周几开始的，如果参数为0，表示一个星期的第一天是从周日开始的；如果参数为1，表示一个星期的第一天是从周一开始的，以此类推。

❑ 在函数的含义对应的表格中，括号中的值表示函数的取值范围。

例10.81 取得当前数据库系统时间的月份、日子和该时间在一年当中是第几个星期。

```
SELECT MONTHNAME(NOW()) AS MONTH ,DAYNAME(NOW()) AS DAY,WEEK(NOW()) AS WEEK
FROM dual
```

这段SQL语句中，是取得当前数据库系统时间的月份、日子和星期值。其中，MONTHNAME函数用来取得当前数据库系统时间的月份，DAYNAME函数用来取得当前数据库系统时间的日子，WEEK函数用来取得当前数据库系统时间在一年当中是第几个星期。其查询结果如下所示。

```
+-----+-----+-----+
| MONTH | DAY   | WEEK |
+-----+-----+-----+
| August | Saturday | 32   |
+-----+-----+-----+
```

从查询的结果中可以看到，根据取得的当前数据库系统的日期时间，MONTHNAME函数返回的值是August，即8月份；DAYNAME函数返回的值是Saturday，即星期六；WEEK函数返回的值是32，即表示当前的系统日期是这一年当中的第32个星期。

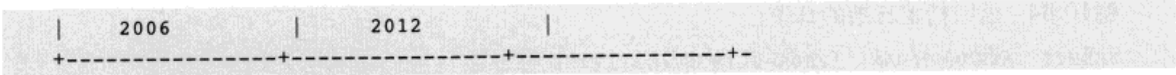
在MySQL数据库中，还可以通过数据库中提供的这些函数，使用加号 (+) 运算符或者是减号 (-) 运算符对取得的日期时间进行一些简单的运算。

例10.82 查询当前数据库系统时间在3年前和3年后分别是哪一年。

```
SELECT YEAR (NOW())-3, YEAR (NOW())+3
FROM dual
```

这段SQL语句中，是查询当前数据库系统时间在3年前和3年后分别是哪一年。其中，YEAR (NOW())-3表示要查询距离数据库当前显示的时间3年前的年份，YEAR (NOW())+3表示要查询距离数据库当前显示的时间3年后的年份。其查询结果如下所示。

```
+-----+-----+
| YEAR (NOW())-3 | YEAR (NOW())+3 |
+-----+-----+
```



从查询的结果中可以看到，根据取得的当前数据库系统的日期时间，YEAR (NOW())-3函数返回的值是2006，YEAR (NOW())+3返回的值是2012。

3. Microsoft SQL Server数据库

在Microsoft SQL Server数据库中，可以使用DATEPART或者DATENAME函数来返回日期指定部分的值。这两个函数的语法规则如下：

```
DATEPART(interval,date)
DATENAME(interval,date)
```

DATEPART函数是以整数的形式返回日期的指定部分的值；DATENAME函数是以字符串的形式返回日期的指定部分的值。其中，参数interval用来指定要返回日期指定部分的值；参数date用来表示指定日期。

参数interval指定的值可以是DAY、MONTH、YEAR等。参数interval指定的值及其在函数中所表示的意义如表10.4所示。

表10.4 参数interval的值及其在函数中所表示的意义

interval的值	缩写	在函数中所表示的意义
YEAR	yy,yyyy	表示指定的年数
MONTH	m,mm	表示指定的月数
DAYOFYEAR	dy,y	表示指定年中的日子
QUARTER	qq,q	表示指定的季度（1个季度为3个月）
WEEK	wk,ww	表示指定一年中的星期数
WEEKDAY (date)	dw,w	表示一周的星期数。其中1代表星期日，7代表星期六
DAY	d,d	表示指定的日子
HOUR	hh,h	表示指定的小时数
MINUTE	mi,n	表示指定的分钟数
SECOND	ss,s	表示指定的秒数
MICROSECOND	mcs	表示指定的微秒数

例10.83 返回指定日期的星期数。

```
SELECT DATEPART(WEEKDAY, '2009-8-15 17:33:37')
```

这段SQL语句中，是取得给定日期的星期数。DATEPART函数中第一个参数WEEKDAY是用来指定取得日期指定部分的值，这里表示取得指定日期的星期值；第二个参数用来表示指定日期值。其查询结果如下所示。

```
DATEPART(WEEKDAY, '2009-8-15 17:33:37')
-----
7
```

从查询结果可以看到，该SQL语句返回的值是7。也就是说，指定的日期2009-8-15 17:33:37中，2009年8月15日这天是星期六。



#### 例10.84 返回指定日期的日子。

```
SELECT DATENAME(DAY, '2009-8-15 17:33:37' )
```

这段SQL语句中，是取得给定日期的日子。DATENAME函数中第一个参数DAY是用来指定取得日期指定部分的值，这里表示取得指定日期中的日子；第二个参数用来表示指定日期值。其查询结果如下所示。

```
DATENAME(WEEKDAY, '2009-8-15 17:33:37' )
```

```
-----  
15
```

从查询结果可以看到，该SQL语句返回的值是15。也就是说，指定的日期2009-8-15 17:33:37中，2009年8月15日这天的日子是15号。

### 10.3.5 取得指定日期所在月的最后一天

在Oracle数据库和MySQL数据库中，如果希望取得指定日期所在月的最后一天，可以使用LAST\_DAY函数。该函数的语法规则如下：

```
LAST_DAY(date)
```

该函数的功能是返回指定日期所在月的最后一天。其中，参数date表示指定的日期。

例10.85 取得当前数据库系统日期所在月的最后一天。

```
SELECT LAST_DAY(SYSTEM)  
FROM dual
```

这段SQL语句是取得当前数据库系统日期所在月的最后一天。其中，LAST\_DAY函数中的参数SYSTEM表示的是当前数据库系统日期时间。其查询结果如下所示。

```
LAST_DAY(SYSTEM)  
-----  
2009-08-31
```

从查询的结果中可以看到，LAST\_DAY函数返回的值是2009-08-31，该值正是当前数据库系统时间所在的月的最后一天对应的日期值。

### 10.3.6 取得两个指定月份的差

在Oracle数据库中，如果希望计算两个日期中月份之间的差，可以使用MONTHS\_BETWEEN函数。该函数的语法格式如下：

```
MONTHS_BETWEEN(date1,date2)
```

该函数的功能是返回指定参数date1和date2对应的日期中相差的月数。其中，参数date1表示第一个日期值，参数date2表示第二个日期值。根据date1和date2的值的不同，该函数返回的结果可以是正数，也可以是负数，还可以是0。

例10.86 计算两个指定日期相差的月数。

```
SELECT MONTHS_BETWEEN(SYSTEM,ADD_MONTH(SYSTEM,-1)) AS BETWEENMONTH  
FROM dual
```

这段SQL语句是用来计算两个指定日期相差的月数。其中MONTHS\_BETWEEN函数中第一个参数SYSTEM表示当前数据库系统的时间，第二个参数使用ADD\_MONTH函数将当前数据库的系统时间对应的月份数减1。其查询结果如下所示。

```
BETWEENMONTH
```

```
1
```

从查询结果可以看出，该函数返回的结果为1。也就是说，两个指定日期之间对应的月数相差为1。如果将ADD\_MONTH函数中第二个参数修改为1，那么显示的结果会是什么呢？读者可以使用Oracle数据库试一下。

### 10.3.7 对日期时间进行舍入操作

在Oracle数据库中，如果希望对指定的日期时间进行四舍五入操作，可以使用ROUND函数。该函数的语法规则如下：

```
ROUND (date[,fmt])
```

该函数的功能是对指定日期时间进行四舍五入操作。其中，参数date表示指定的日期时间，参数fmt表示日期时间四舍五入的方式。如果fmt的值表示年份，则以7月1日作为舍入的分界点；如果fmt的值表示月份，则以16日作为舍入的分界点；如果fmt的值表示日子，则以12:00AM作为舍入的分界点。其中，参数fmt是可选的，如果不指定fmt，则默认返回距离指定日期最近的日期。

例10.87 对指定日期时间进行舍入操作。

```
SELECT SYSTEM,ROUND(SYSTEM,'YYYY')
FROM dual
```

这段SQL语句是用来对指定日期时间进行舍入操作。其中SYSTEM表示当前系统的日期时间，ROUND函数中，第一个参数表示的是当前系统的日期时间，第二个参数表示的是按照年份对当前系统的日期时间进行四舍五入的操作。其查询结果如下所示。

```
SYSTEM          ROUND(SYSTEM,'YYYY')
-----
2009-8-15 23:17:47  2010-01-01 00:00:00
```

从查询的结果可以看到，当前数据库的系统时间是2009-8-15 23:17:47，使用ROUND函数按照年份进行四舍五入操作之后，其显示的时间变为2010-01-01 00:00:00。因为，在Oracle数据库中，按照年份进行四舍五入操作，是以7月1日作为舍入的分界点，因为当前日期时间已经超过了7月1日，所以返回的日期时间就是下一年1月1日零点对应的的时间。

当然，ROUND函数的第二个参数中还可以是MONTH、DD、DAY、HH24、HH12等其他的日期时间值，读者可以在Oracle数据库中修改ROUND函数中的参数fmt的值，看一下其查询结果有什么变化。由于篇幅所限这里就不再一一举例了。

### 10.3.8 截断指定的日期时间

在Oracle数据库中，如果希望对指定的日期时间进行截断操作，可以使用TRUNC函数。该函数的语法规则如下：

TRUNC(date[,fmt])

该函数的功能是截断指定的日期时间。其中参数date用来指定日期时间，参数fmt表示日期时间的截断方式。如果fmt的值表示年份，则返回的结果是当前年的1月1日；如果fmt的值表示月份，则返回的结果为本月的1日。

例10.88 利用TRUNC函数得到当前日期所在月份的第一天。

```
SELECT SYSTEM,TRUNC(SYSTEM,'MM')
FROM dual
```

这段SQL语句是利用TRUNC函数得到当前日期所在月份的第一天。其中SYSTEM表示当前系统的日期时间，TRUNC函数中，第一个参数表示的是当前系统的日期时间，第二个参数表示的是按照月份对当前系统的日期时间进行截取的操作。其查询结果如下所示。

SYSTEM	TRUNC(SYSTEM,'MM')
2009-8-15 23:32:27	2009-08-01 00:00:00

从查询的结果可以看到，当前数据库的系统时间是2009-8-15 23:32:27，使用TRUNC函数按照月份对当前系统的日期时间进行截取操作，其显示的时间变为2009-08-01 00:00:00。该时间是，当前数据库的系统时间所在月的第一天。因为，在Oracle数据库中，按照月份对当前系统的日期时间进行截取的操作，TRUNC函数返回结果值是当前月的1日。

当然，TRUNC函数的第二个参数中还可以是MONTH、DD、DAY、HH24、HH12等其他的日期时间值，读者可以在Oracle数据库中修改TRUNC函数中的参数fmt的值，看一下其查询结果有什么变化。由于篇幅所限这里就不再一一举例了。

**注意** 在Microsoft SQL Server数据库中，可以使用CONVERT函数对指定日期时间进行截断操作。其具体的使用方法可以参看10.4.2小节。

## 10.4 类型转换函数

在实际的开发过程中，有时需要将数值从一种数据类型转换到另一种数据类型。例如将日期类型转换为字符类型，将字符串转换成一个日期类型的数据，将数字转换成字符类型的值等。这个时候就需要用到类型转换函数。这里以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，介绍几种主要的类型转换函数。

### 10.4.1 字符转换函数

不同的数据库对字符转换函数的处理也不完全相同。这里以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，介绍在这3种数据库中使用字符转换函数的方法。

#### 1. Oracle数据库

在Oracle数据库中，如果希望将一个数字值或者是一个日期值转换为字符类型的数据，可以使用TO\_CHAR函数。该函数既可以处理数字值，也可以用来处理日期值。

首先来看TO\_CHAR函数转换数字值。使用TO\_CHAR函数可以将数字值转换为字符类型的数据。



其语法格式如下：

```
TO_CHAR(n[,fmt[, 'nlsparams']])
```

该函数的功能是将数字值转换为字符类型的数据。其中，参数n表示要转换为字符类型的数据的数字值；参数fmt用来指定数字值转换字符类型的数据的方式。参数nlsparams表示数据库中的语言信息，一般很少用。参数fmt和参数nlsparams是可选的。

**例10.89** 将教师信息表中的教师工资前加上一个本地货币的符号。

```
SELECT teaID ,teaName , dept ,salsry,TO_CHAR(salary,'L9999.99')
FROM T_teacher
```

这段SQL语句中，是在教师信息表中，将显示的教师工资前加上一个本地货币的符号。这里使用TO\_CHAR函数将教师工资转换为字符类型的数据。其中，第一个参数salary表示教师的工资，第二个参数L9999.99中，字符L表示在教师工资的前面还要多一个本地货币符号，数字9999.99表示教师的工资要显示到小数点后两位。其查询结果如下所示。

teaID	teaName	dept	salary	TO_CHAR(salary,'L9999.99')
t102225	赵伟	计算机系	3000	¥ 3000.00
t103265	张昌	计算机系	3800	¥ 3800.00
t105320	于波	计算机系	2800	¥ 2800.00
t106358	毛翠	计算机系	4000	¥ 4000.00
t156354	王新	数学系	2500	¥ 2500.00
t156355	李中	数学系	4200	¥ 4200.00
t181585	李慧	物理系	3500	¥ 3500.00
t186585	孙立	物理系	3200	¥ 3200.00

从查询的结果可以看到，在使用了TO\_CHAR函数之后，其显示的教师工资的前面多了一个本地货币符号，并且工资显示到了小数点后两位。

TO\_CHAR函数中第二个参数除了可以在指定数字的前面加上一个本地货币符号并指定数字值显示的位数之外，还有其他的数据转换方式。其符号以及代表的转换方式如表10.5所示。

表10.5 TO\_CHAR函数数字值数据转换方式的符号及其意义

符 号	意 义
9	表示数字。如果转换的数字值前面有0，则0被忽略
0	表示数字补0。如果转换的数字值的位数不足，则会强制补0
L	表示本地货币符号。在转换后的数字值前加上一个本地货币符号
C	表示国际货币符号。在转换后的数字值前加上一个国际货币符号
\$	表示美元符号。在转换后的数字值前加上一个美元符号
D	在转换后的数字值的指定位置显示小数点
G	在转换后的数字值的指定位置显示分隔符
.	表示小数点。在转换后的数字值的指定位置显示小数点
,	表示千位分隔符。在转换后的数字值的指定位置显示分隔符

在表10.5中列出了TO\_CHAR函数数字值数据转换方式的符号及其意义。这样就不难理解例10.89显示的查询结果了。读者可以考虑一下，在教师信息表中，如果现在希望使用千位分隔符将教师工资显

示出来，使用TO\_CHAR函数应该如何完成。

**提示** 教师显示的工资形式是3,000。即显示的数字值中每隔3位就用一个逗号将其分开。

TO\_CHAR函数不仅可以数字值转换为字符类型的数据，还可以将日期类型的数据转换为字符类型的数据。将日期类型的数据转换为字符类型数据的语法格式如下：

TO\_CHAR(date,fmt)

该函数的功能是将日期类型的值转换为字符类型的数据。其中，参数date表示要转换为字符类型的数据的日期值；参数fmt用来指定日期值转换字符类型的数据的方式。

**例10.90** 将学生信息表中学生的出生日期以YYYY/MM/DD的形式显示。

```
SELECT stuID, stuName, age,sex,TO_CHAR(birth,'YYYY/MM/DD')
FROM T_student
```

这段SQL语句是将学生信息表中学生的出生日期以指定的形式显示。这里使用TO\_CHAR函数将学生的出生日期转换为字符类型的数据。其中，TO\_CHAR函数中第一个参数表示学生的出生日期，第二个参数表示要将学生的出生日期转换为YYYY/MM/DD的形式显示。其查询结果如下所示。

stuID	stuName	age	sex	TO_CHAR(birth,'YYYY/MM/DD')
s102203	赵亮	23	男	1986/05/16
s112303	郑茹	21	女	1988/01/25
s115263	王海	23	男	1986/08/02
s206363	张明	22	男	1987/04/23
s221256	王昌鹤	24	男	1985/03/18
s231456	王玉梅	22	女	1987/03/28
s232516	李玉峰	24	女	1985/08/16
s253263	李凤	24	女	1985/06/06

从查询结果可以看到，通过使用TO\_CHAR函数，学生信息表中学生的出生日期都是以YYYY/MM/DD的形式显示出来的。

TO\_CHAR函数中除了可以将日期类型转换为YYYY/MM/DD的形式之外，还有其他的数据转换方式。其符号以及代表的转换方式如表10.6所示。

表10.6 TO\_CHAR函数日期值数据转换方式的符号及其意义

符 号	意 义
YY	表示年份。以2位数字的形式显示。例如，03、09等
YYYY	表示年份。以4位数字的形式显示
MM	表示月份。以2位数字的形式显示
WW	表示一年当中的星期数。显示范围1~53
W	表示该月当中的星期数。显示范围1~5
DD	表示日子。以2位数字的形式显示
YEAR	表示年份。以英文的形式显示
MONTH	表示月份。以英文的全拼的形式显示。例如，MARCH
MON	表示月份。以英文的缩写形式显示。例如，MAR

(续)

符 号	意 义
DAY	表示工作日。以英文的全拼的形式显示。例如，SUNDAY
DY	表示工作日。以英文的全拼的形式显示。例如，SUN
HH12	表示时间。以12小时的形式显示。例如，下午4点表示为04
HH24	表示时间。以24小时的形式显示。显示范围0~23。例如，下午4点表示为16
MI	表示分钟。显示范围0~59
SS	表示秒。显示范围0~59
Q	表示季度。显示范围1~4

在表10.6中列出了TO\_CHAR函数日期值数据转换方式的符号及其意义。这样就不难理解例10.90显示的查询结果了。读者可以考虑一下，在学生信息表中，如果现在希望学生的出生日期以MAY 16 1986这样的形式显示出来，使用TO\_CHAR函数应该如何完成。

2. MySQL数据库

在MySQL数据库中，如果想将一个数字值或者是一个日期值转换为字符类型的数据，可以使用CAST函数。该函数的语法规则如下：

```
CAST(expression AS datatype[length])
```

该函数的功能是将expression中的值转换为指定的数据类型。其中，参数expression表示要转换为指定数据类型的表达式；参数datatype表示要转换的数据类型，该数据类型可以使用字符类型、数值类型、日期类型等；参数length用来指定指定数据类型数据的长度，它是可选的。

例10.91 将教师信息表中教师的工资信息转换为字符类型。

```
SELECT teaID,teaName, CAST(salary AS CHAR (5))
FROM T_teacher
```

这段SQL语句是查询教师信息表中教师的工资信息并将取得的教师工资转换为字符类型。在CAST函数中，salary表示教师工资的列，CHAR用来指定将教师工资转换为字符类型的数据，并指定了字符类型数据的长度。其查询结果如下所示。

teaID	teaName	salary
t102225	赵伟	3000
t103265	张昌	3800
t105320	于波	2800
t106358	毛翠	4000
t156354	王新	2500
t156355	李中	4200
t181585	李慧	3500
t186585	孙立	3200

**注意** CAST函数除了可以将指定的数据类型转换成字符类型的数据之外，也可以在连接不同数据类型时对其进行数据类型转换，根据CAST函数中指定的参数不同，可以将其转换为数值类型、日期类型等。

### 3. Microsoft SQL Server数据库

在Microsoft SQL Server数据库中，也可以使用CAST函数进行数据类型的转换，其使用方法与MySQL数据库中使用CAST函数的方法相同，这里就不再举例了。

除了CAST函数之外，如果希望将一个数字类型的数据转换为字符类型的数据，在Microsoft SQL Server数据库中还可以使用STR函数。该函数的语法规则如下：

```
STR (float[,length [,decima]])
```

该函数的功能是将数字类型的数据转换为字符类型的数据。其中，参数float指定的是浮点类型的数据；参数length表示要返回的字符串的长度，该长度包括小数点、数字、符号或者空格所占的位数，其默认的长度为10；参数decima指定小数点右边小数的位数。参数length和参数decima的值都应该是正数，它们是可选的。当在默认情况下或者小数参数为0时，会将指定数字值四舍五入为整数。

**注意** STR函数中参数length的值应该大于或者等于小数点左边的数字和数字符号的位数。如果length的值小于小数点左边的数字和数字符号的位数，则STR函数会返回“\*”号。

例10.92 将指定的数字转换为字符类型。

```
SELECT STR(12345.67,3), STR(12345.67,5), STR(12345.67,8,2)
```

这段SQL语句是将指定的数字12345.67转换为字符类型的数据。其中，第一个STR函数中，由于第二个参数指定长度比指定的浮点类型的数字12345.67小，所以该函数的返回值会是三个“\*\*\*”号，表示数据溢出；第二个STR函数中，指定返回字符串的长度为5；第三个STR函数中，指定了要返回的字符串长度为7，并指定小数点右边小数的位数为2。其查询结果如下所示。

```
STR(12345.67,3)  STR(12345.67,5)  STR(12345.67,8,2)
-----
***              12346             12345.67
```

### 10.4.2 日期转换函数

不同的数据库对日期值进行转换的函数也不完全相同。这里以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例，介绍在这3种数据库中使用日期转换函数的方法。

#### 1. Oracle数据库

在Oracle数据库中，如果希望字符串转换为日期类型的数据，可以使用TO\_DATE函数。该函数的语法格式如下：

```
TO_DATE(string[,fmt[,nlsparams]])
```

该函数的功能是将字符串转换为日期类型的数据。其中，参数string表示要转换为日期类型的数据的字符串；参数fmt用来指定字符串转换日期类型的数据的方式；参数nlsparams表示数据库中的语言信

息，一般很少用。参数fmt和参数nlsparams是可选的。

TO\_DATE函数中有许多将指定字符串转换为日期类型的数据的方式。其符号以及代表的转换方式如表10.7所示。

表10.7 TO\_DATE函数字符串数据转换方式的符号及其意义

符 号	意 义
YY	表示年份。以2位数字的形式显示。例如，03、09等
YYYY	表示年份。以4位数字的形式显示
MM	表示月份。以2位数字的形式显示
WW	表示一年当中的星期数。显示范围1~53
W	表示该月当中的星期数。显示范围1~5
DD	表示日子。以2位数字的形式显示
YEAR	表示年份。以英文的形式显示
MONTH	表示月份。以英文的全拼的形式显示。例如，MARCH
MON	表示月份。以英文的缩写的形式显示。例如，MAR
DAY	表示工作日。以英文的全拼的形式显示。例如，SUNDAY
DY	表示工作日。以英文的全拼的形式显示。例如，SUN
HH12	表示时间。以12小时的形式显示。例如，下午4点表示为04
HH24	表示时间。以24小时的形式显示。显示范围0~23。例如，下午4点表示为16
MI	表示分钟。显示范围0~59
SS	表示秒。显示范围0~59
Q	表示季度。显示范围1~4

在表10.7中列出了TO\_DATE函数字符串转换为日期数据的方式。下面来看两个使用TO\_DATE函数将字符串转换成日期类型数据的例子。

例10.93 将当前数据库系统时间以YYYYMMDD的形式显示。

```
SELECT SYSTEM,TO_DATE(TO_CHAR(SYSTEM,'YYYYMMDD'),'YYYYMMDD') AS NEWDATE
FROM dual
```

这段SQL语句中，是取得当前数据库的系统时间，并将该时间以YYYYMMDD的形式显示。使用SYSTEM函数取得当前系统的日期时间值，使用TO\_DATE函数将当前系统日期时间以YYYYMMDD的形式显示。由于TO\_DATE函数中第一个参数应该是一个字符串，所以，这里需要使用TO\_CHAR函数先将当前系统日期时间转换为字符类型的数据。其查询结果如下所示。

SYSTEM	NEWDATE
2009-8-16 14:40:30	20090816

从查询的结果中可以看到，结果中的日期值显示的是20090816，即日期值是以YYYYMMDD的形式显示的。

例10.94 显示指定日期是星期几。

```
SELECT TO_CHAR(TO_DATE('20090816','YYYYMMDD'),'DAY','NLS_DATE_LANGUAGE = American') AS DAY
FROM dual
```



## 零基础学SQL

这段SQL语句中，是用来显示指定日期是星期几，并将显示的值以英文全拼的形式显示。在TO\_DATE函数中，第一个参数是一个字符串，指定了一个日期值。第二个参数用来指定日期值的转换方式。并将使用TO\_DATE函数转换的日期值作为TO\_CHAR函数的第一个参数，TO\_CHAR函数的第二个参数用来指定显示该日期对应的星期值，第三个参数用来设置数据库中的日期语言，将该日期语言设置为英文。其查询结果如下所示。

```
DAY
-----
Sunday
```

从查询的结果中可以看到，结果中显示的是Sunday，即星期日。从例10.93中可以了解到当前的系统日期是2009年8月16日，这一天正好是星期日。

### 2. MySQL数据库

在MySQL数据库中，如果希望对日期时间值进行格式转换，可以使用DATE\_FORMAT函数。该函数的语法格式如下：

```
DATE_FORMAT(date,fmt)
```

该函数的功能是根据参数fmt给定的格式对日期时间值进行格式转换。其中，参数date表示指定的日期时间；参数fmt用来指定字符串转换日期类型的数据的方式。

例10.95 将当前数据库系统日期以指定的格式显示出来。

```
select DATE_FORMAT(NOW(), '%W %M%D%Y ')
from dual
```

这段SQL语句是使用DATE\_FORMAT函数将当前系统日期以指定的格式显示出来。在DATE\_FORMAT函数中，第一个参数表示当前数据库系统的日期时间，第二个参数用来指定日期时间的显示格式。其中，%W显示当前数据库系统日期中时间是星期几，%M表示显示当前数据库系统日期中的月份，%D表示显示当前数据库系统日期中的日子，%Y表示显示当前数据库系统日期中的年份。其查询结果如下所示。

```
+-----+
| DATE_FORMAT(NOW(), '%M%D%Y%W ') |
+-----+
| Sunday August 16th 2009          |
+-----+
```

从查询结果可以看到，结果中是以英文的形式将当前数据库系统日期时间中的星期、月份、日子以及年份显示的。在MySQL数据库中，不管其日期语言设置为什么，其星期、月份、日子、年份等的名字总是以英文的形式给出。

DATE\_FORMAT函数除了可以处理日期，还可以处理时间。下面来看一个DATE\_FORMAT函数处理时间的例子。

例10.96 将当前数据库系统时间以指定的格式显示出来。

```
select NOW(),DATE_FORMAT(NOW(), '%H:%i:%s') AS TIME
from dual
```

这段SQL语句是使用DATE\_FORMAT函数将当前系统时间以指定的格式显示出来。其中，NOW函

数表示系统当前的日期时间。在DATE\_FORMAT函数中，第一个参数表示当前数据库系统的日期时间，第二个参数用来指定日期时间的显示格式。其中，%H显示当前数据库系统时间中的小时，%i表示显示当前数据库系统时间中的分钟，%s表示显示当前数据库系统时间中的秒%。其查询结果如下所示。

+	-----+	+	-----+	+	-----+
	NOW ()		TIME		
+	-----+	+	-----+	+	-----+
	2009-08-16 15:53:59		15:53:59		
+	-----+	+	-----+	+	-----+

从查询结果可以看到，表示时间的TIME字段是将当前数据库系统时间以时:分:秒的形式显示出来的。

DATE\_FORMAT函数中除了可以使用%W、%M、%D、%Y、%H、%i、%s之外，还有其他的数据转换方式。其符号以及代表的转换方式如表10.8所示。

表10.8 DATE\_FORMAT函数日期时间转换方式的符号及其意义

符 号	意 义
%Y	表示年份。以4位数字的形式显示。例如，2008、2009
%y	表示年份。以2位数字的形式显示。例如，08、09
%M	表示月份。以英文的形式显示。例如，January
%b	表示月份。以英文缩写的形式显示。例如，Jan
%m	表示月份。以2位数字形式显示。例如，01、02
%c	表示月份。以数字形式显示。例如，1、2
%W	表示星期。以英文的形式显示。例如，Monday
%a	表示星期。以英文缩写的形式显示。例如，Mon
%w	表示星期。以数字的形式显示。0表示星期日，6表示星期六
%D	表示日子。以英文后缀的形式显示。例如，1st、2nd
%d	表示日子。以2位数字的形式显示。例如，01、02
%e	表示日子。以数字的形式显示。例如，1、2
%U	表示一年当中的星期数。以星期日作为首日。显示范围00~52
%u	表示一年当中的星期数。以星期一作为首日。显示范围00~52
%j	表示一年中的天数。以3位数字的形式显示。例如，001
%H	表示小时。以2位数字的形式显示。例如，01、02
%k	表示小时。以24小时制的形式显示
%l	表示小时。以12小时制的形式显示
%I或者%h	表示小时。以2位数字12小时制的形式显示
%i	表示分钟。以2位数字的形式显示
%S或者%s	表示秒。以2位数字的形式显示
%p	表示上午或者下午。其中，AM表示上午，PM表示下午
%T	表示24小制的时间。显示范围00:00:00~23:59:59
%r	表示12小制的时间。显示范围12:00:00AM~11:59:59PM
%%	表示标示符%

零基础学SQL

在表10.8中列出了DATE\_FORMAT函数日期时间转换方式的符号及其意义。这样就不难理解例10.94和例10.95显示的查询结果了。读者可以在MySQL数据库中使用上述的符号将日期时间转换为自己想要的格式。

DATE\_FORMAT函数既可以处理日期值，也可以处理时间值。在MySQL数据库中，除了可以使用DATE\_FORMAT函数处理时间外，还可以使用TIME\_FORMAT函数对时间进行处理。该函数的语法格式如下：

```
TIME_FORMAT(time,fmt)
```

该函数的功能是根据参数fmt给定的格式对指定时间值进行格式转换。其中，参数time表示指定的时间；参数fmt用来指定字符串转换日期类型的数据的方式。

TIME\_FORMAT函数的使用方法与DATE\_FORMAT()函数的使用方法相同，但是该函数只能用来处理时间值，而不能用来处理日期值。

3. Microsoft SQL Server数据库

在Microsoft SQL Server数据库中，如果希望对日期时间值进行格式转换，可以使用CONVERT函数。该函数的语法格式如下：

```
CONVERT (datatype[length],date[,style])
```

该函数的功能是对日期时间值进行格式转换。其中，参数datatype表示指定日期时间转换后的数据类型，在数据类型中可以使用length指定其长度，参数length是可选的；参数date指定需要转换的日期时间；参数style表示日期的转换形式，该参数也是可选的。

在CONVERT函数中参数style取不同的值，其日期时间显示的格式是不相同的。参数style主要的值与其对应的日期时间显示格式如表10.9所示。

表10.9 参数style主要的值与其对应的日期时间显示格式

style值	标 准	输出格式
0	默认	mon dd yyyy hh:miAM（或者PM）
1	美国	mm/dd/yy
2	ANSI	yy.mm.dd
3	英国/法国	dd/mm/yy
4	德国	dd.mm.yy
5	意大利	dd-mmdyy
6	—	dd mon yy
7	—	dd mon,yy
8	—	hh:mm:ss
9	默认毫秒	mon dd yyyy hh:mi:ss:mmmAM（或者PM）
10	美国	mm-dd-yy
11	日本	yy/mm/dd
12	ISO	yymmdd
13	欧洲默认	fd mon yyyy hh:mm:ss:mmm（24小时制）
14	—	hh:mi:ss:mmm（24小时制）

- ❑ 如果希望显示的年份是以世纪的形式显示，可以将style值加上100。例如，style值为1时，显示的日期格式为08/16/09；style值为101时，显示的日期格式为08/16/2009。这种情况适用于表中的所有style值。
  - ❑ style值为0、9或者13时，其显示日期时间格式中的年都是以四位的形式给出。
  - ❑ style值为0、7或者13时，其显示日期时间格式中的月份是以3位字符的形式给出。例如，1月January，其显示的格式为Jan。
  - ❑ style值为13或者14时，其返回的日期时间格式中，时间是以24小时制的形式给出。
- 例10.97** 将指定日期时间按照指定格式转换。

```
SELECT CONVERT (VARCHAR (12), GETDATE(), 11), CONVERT (VARCHAR (12), GETDATE(), 111)
```

这段SQL语句是将取得的当前系统日期时间转换为参数style所指定的日期格式来显示。在CONVERT函数中，参数VARCHAR表示将指定日期时间转换为字符类型的数据；GETDATE函数表示取得当前系统日期时间；参数11和参数111表示指定日期时间的显示格式。其查询结果如下所示。

```
CONVERT (VARCHAR (12), GETDATE(), 11)    CONVERT (VARCHAR (12), GETDATE(), 111)
-----
09/08/16                                2009/08/16
```

从查询结果中，可以看到参数style中的值没加100和加上100以后显示结果的不同。在CONVERT函数中，当参数style的值为11时，其显示的年为09；当参数style的值为111时，其显示的年则变为2009。

### 10.4.3 数值转换函数

在Oracle 9i及其以后的版本中，新增加了数值转换函数TO\_NUMBER。TO\_NUMBER函数的语法规则如下：

```
TO_NUMBER(char[,fmt[,nlsparams]])
```

该函数的功能是将字符数据转换成指定格式的数值类型的数据。其中，参数char表示要转换成数值类型的字符，参数fmt用来指定字符转换数值类型的数据的方式。参数nlsparams表示数据库中的语言信息，一般很少用。参数fmt和参数nlsparams是可选的。

**例10.98** 将指定的字符数据转换为数值类型的数据。

```
SELECT TO_NUMBER('$3000', 'L9999D99')
FROM dual
```

这段SQL语句是将字符'\$3000'转换为数值类型的数据。其中，参数L9999D99中，字符L表示在教师工资的前面还要多一个本地货币符号，数字9999D99表示转换的数值要显示到小数点后两位，字符D表示小数点（参看表10.5）。其查询结果如下所示。

```
TO_NUMBER('$3000', 'L9999D99')
-----
¥3000.00
```

从查询结果可以看到，在使用了TO\_NUMBER函数之后，显示的数值显示到了小数点后两位并返回本地的货币符号。

## 10.5 比较函数

在实际的开发应用中，有些时候开发人员或者用户对集合中最大值或者集合中的最小值感兴趣，希望从中选择其中值最大的一个或者值最小的一个，这个时候就需要对集合中数字进行比较；还有些时候开发人员或者用户关心两个字符串的排序大小，这个时候就需要对字符串的排序大小进行比较。在Oracle和MySQL数据库中提供了获取集合中最大值或者最小值的函数，MySQL数据库中还提供了对两个字符串进行比较的函数。这一节就来介绍Oracle和MySQL数据库中的几个比较函数。

### 10.5.1 求集合中的最小值

在SQL语句中，如果希望取得给定集合中的最小值，在Oracle数据库和MySQL数据库中使用LEAST函数。该函数的语法格式如下：

```
LEAST(n1,n2,n3...)
```

该函数的功能是取得给定集合中的最小值。其中，参数n1表示集合中的第一个数字；参数n2表示集合中的第二个数字；参数n3表示集合中的第三个数字。在这个函数中，可以指定任意多个数字。如果其中有一个参数的值为NULL，则该函数的返回值就为NULL。

例10.99 求给定集合中的最小值。

```
SELECT LEAST(10,15,6,20)
FROM dual
```

这段SQL语句是在指定集合中的这些数字中，选取其中值最小的一个。其查询的结果如下所示。

```
LEAST(10,15,6,20)
```

```
-----
6
```

从查询结果可以看到，LEAST(10,15,6,20)的返回值是6。在给定集合的这4个值中，数值6是其中最小的一个。

**注意** Microsoft SQL Server数据库中没有求集合最小值的函数。

### 10.5.2 求集合中的最大值

在SQL语句中，如果希望取得给定集合中的最大值，在Oracle数据库和MySQL数据库中使用GREATEST函数。该函数的语法格式如下：

```
GREATEST(n1,n2,n3...)
```

该函数的功能是取得给定集合中的最大值。其中，参数n1表示集合中的第一个数字；参数n2表示集合中的第二个数字；参数n3表示集合中的第三个数字。在这个函数中，可以指定任意多个数字。如果其中有一个参数的值为NULL，则该函数的返回值就为NULL。

例10.100 求给定集合中的最大值。

```
SELECT GREATEST(10,15,6,20)
FROM dual
```

这段SQL语句是在指定集合中的这些数字中，选取其中值最大的一个。其查询的结果如下所示。



```
GREATEST(10,15,6,20)
```

```
-----  
20
```

从查询结果可以看到，GREATEST(10,15,6,20)的返回值是20。在给定集合的这4个值中，数值20是其中最大的一个。

**注意** Microsoft SQL Server数据库中没有求集合最大值的函数。

### 10.5.3 比较两个字符串

在SQL语句中，如果希望对两个字符串进行比较，在MySQL数据库中使用STRCMP函数。该函数的语法格式如下：

```
STRCMP(string1,string2)
```

该函数的功能是对指定的两个字符串string1和string2进行比较。其中，参数string1表示要进行比较的第一个字符串；参数string2表示要进行比较的第二个字符串。如果string1= string2，函数返回值为0；如果string1> string2，函数返回值为1；如果string1<string2，函数返回值为-1。在MySQL 4.0版本之后，默认情况下，STRCMP函数不区分字母的大小写。如果其中有一个参数的值为NULL，则该函数的返回值就为NULL。

**例10.101** 比较两个字符串。

```
select STRCMP('mysql','mysql5.0'),STRCMP('mysql','mySQL'),STRCMP('mysql','my')  
FROM dual
```

这段SQL语句是使用STRCMP函数对指定的字符串进行比较，并返回比较的结果。其查询的结果如下所示。

```
STRCMP('mysql','mysql5.0')  STRCMP('mysql','mySQL')  STRCMP('mysql','my')  
-----  
-1                          0                          1
```

从查询结果可以看到，STRCMP函数会根据两个字符串比较的结果，返回值-1、0或者1，并且该函数在默认的情况下，并不区分字符串中字母的大小写。

**注意** Oracle数据库和Microsoft SQL Server数据库不支持STRCMP函数。

## 10.6 空值处理函数

在实际的开发应用中，对数据库进行查询操作时经常会遇到空值NULL，很多时候需要对NULL进行处理。在数据库中，一般都提供了对空值的处理函数。这一节就来介绍数据库中几种常用的对空值进行处理的函数。

### 10.6.1 NVL函数与IFNULL函数

NVL函数是在Oracle数据库中用来处理NULL值的函数。该函数的语法规则如下：

## 零基础学SQL

**NVL(expression1,expression2)**

该函数的功能是将其中的NULL值转化为一个实际的值。其中，参数expression1和参数expression2可以是相互匹配的任意数据类型。如果expression1的值为NULL，则函数将返回expression2对应的值；如果expression1的值不为NULL，则函数将返回expression1对应的值。

例10.102 查询教师工资和津贴的总收入。

```
SELECT teaID ,teaName ,salary,pension, NVL(salary+pension,salary)AS income
FROM T_teacher
```

这段SQL语句是查询教师工资和津贴的总收入。这里在计算教师工资和津贴的总收入时，使用了NVL函数处理字段pension中的NULL值。其查询结果如下所示。

teaID	teaName	salary	pension	income
t102225	赵伟	3000	260.5	3260.5
t103265	张昌	3800	300	4100
t105320	于波	2800		2800
t106358	毛翠	4000	289.3	4289.3
t156354	王新	2500		2500
t156355	李中	4200	310.2	4510.2
t181585	李慧	3500	278	3778
t186585	孙立	3200		3200

查询结果中，表示教师津贴的字段pension中存在空值（NULL值），即不是每个教师都有津贴。字段income表示的是教师工资和津贴的总收入。在字段income显示的结果中可以看到，如果教师有津贴，则教师的总收入就是教师工资和津贴的加和；如果教师没有津贴，则教师的总收入就是字段salary所表示的教师工资的值。在表示教师总收入的字段income中，不存在NULL值。

在Microsoft SQL Server数据库和MySQL数据库中，可以使用IFNULL函数来处理NULL值。该函数的语法规则如下：

**IFNULL(expression1,expression2)**

该函数的功能与Oracle数据库中NVL函数的功能相同，也是将其中的NULL值转化为一个实际的值。其中，参数expression1和参数expression2可以是相互匹配的任意数据类型。如果expression1的值为NULL，则函数将返回expression2对应的值；如果expression1的值不为NULL，则函数将返回expression1对应的值。

对于例10.102，使用IFNULL函数的SQL语句可以写成如下的形式。

```
SELECT teaID ,teaName ,salary,pension, IFNULL(salary+pension,salary)AS income
FROM T_teacher
```

这段SQL语句是查询教师工资和津贴的总收入。这里在计算教师工资和津贴的总收入时，使用了IFNULL函数处理字段pension中的NULL值。其查询结果如下所示。

teaID	teaName	salary	pension	income
t102225	赵伟	3000	260.5	3260.5
t103265	张昌	3800	300	4100

t105320	于波	2800	NULL	2800
t106358	毛翠	4000	289.3	4289.3
t156354	王新	2500	NULL	2500
t156355	李中	4200	310.2	4510.2
t181585	李慧	3500	278	3778
t186585	孙立	3200	NULL	3200

8 rows in set

这里的查询结果与例10.102中使用NVL函数查询的结果相同。

**注意** 在Oracle数据库中可以使用NVL函数处理NULL值，在Microsoft SQL Server数据库和MySQL数据库中，可以使用IFNULL函数来处理NULL值。

### 10.6.2 NVL2函数

在Oracle 9i及其以后的版本中，除了在10.6.1节中讲到的NVL函数之外，还可以使用NVL2函数。该函数的语法规则如下：

**NVL2(expression1,expression2,expression3)**

该函数的功能也是对NULL值进行处理的。其中，参数expression1可以是任意的数据类型；参数expression2和参数expression3是除了LONG类型以外的其他任意数据类型。如果expression1的值不为NULL，则函数将返回expression2对应的值；如果expression1的值为NULL，则函数将返回expression3对应的值。

**例10.103** 查询教师工资和津贴的总收入（使用NVL2函数）。

```
SELECT teaID ,teaName ,salary,pension, NVL2(pension ,salary+pension,salary)AS income
FROM T_teacher
```

这段SQL语句是查询教师工资和津贴的总收入。这里在计算教师工资和津贴的总收入时，使用了NVL2函数处理字段pension中的NULL值。NVL2函数需要接收3个参数。其查询结果如下所示。

teaID	teaName	salary	pension	income
t102225	赵伟	3000	260.5	3260.5
t103265	张昌	3800	300	4100
t105320	于波	2800		2800
t106358	毛翠	4000	289.3	4289.3
t156354	王新	2500		2500
t156355	李中	4200	310.2	4510.2
t181585	李慧	3500	278	3778
t186585	孙立	3200		3200

查询结果中，表示教师津贴的字段pension中存在空值（NULL值），即不是每个教师都有津贴。字段income表示的是教师工资和津贴的总收入。这里，如果表示教师津贴的字段pension为空值（NULL值），则显示字段salary与字段pension加和的值；如果字段pension不为空值（NULL值），则显示字段salary中的值。

### 10.6.3 ISNULL函数

在MySQL数据库和Microsoft SQL Server数据库中，可以使用ISNULL函数对查询结果中是否为NULL值进行判断。该函数的语法规则如下：

**ISNULL(expression)**

该函数的功能是根据expression的值是否为NULL，返回值0或者1。如果expression的值不为NULL，则该函数返回0；如果expression的值为NULL，则该函数返回1。

**例10.104** 查询教师津贴，并判断津贴是否为NULL。

```
SELECT teaID ,teaName ,salary,pension,ISNULL(pension)
FROM T_teacher
```

这段SQL语句是对查询的教师津贴值是否为NULL进行判断。这里在表示教师津贴的字段pension后面加了一个ISNULL函数，使用该函数对字段pension是否为NULL值进行判断，并根据判断结果返回值0或者1。其查询结果如下所示。

teaID	teaName	salary	pension	ISNULL(pension)
t102225	赵伟	3000	260.5	0
t103265	张昌	3800	300	0
t105320	于波	2800	NULL	1
t106358	毛翠	4000	289.3	0
t156354	王新	2500	NULL	1
t156355	李中	4200	310.2	0
t181585	李慧	3500	278	0
t186585	孙立	3200	NULL	1

8 rows in set

从查询结果中可以看到，表示教师津贴的字段pension对应的值如果为NULL，则ISNULL(pension)函数的值为1；如果表示教师津贴的字段pension对应的值不为NULL，则ISNULL(pension)函数的值就为0。

### 10.6.4 COALESCE函数

在Oracle数据库和MySQL数据库中，还可以使用COALESCE函数对NULL值进行处理。该函数的语法规则如下：

**COALESCE(expression1,expression2,expression3,...)**

该函数的功能是返回表达式中第一个值不为NULL的表达式。也就是说，在给定的这些表达式中，如果参数expression1的值不为NULL，则该函数返回参数expression1的值；如果参数expression1的值为NULL，而参数expression2的值不为NULL，则该函数返回参数expression2的值；如果参数expression1的值和参数expression2的值都为NULL，而参数expression3的值不为NULL，则该函数返回参数expression3的值。以此类推，直到找到第一个值不为NULL的表达式，并将该表达式对应的值返回。

例10.105 以查询教师的工资和津贴为例，验证COALESCE函数。

```
SELECT pension,salary,COALESCE(pension,salary)
FROM T_teacher
```

这段SQL语句是以字段pension和字段salary的值作对比，通过字段pension和字段salary的值来验证COALESCE函数。其查询结果如下所示。

pension	salary	COALESCE(pension,salary)
260.5	3000	260.5
300	3800	300
NULL	2800	2800
289.3	4000	289.3
NULL	2500	2500
310.2	4200	310.2
278	3500	278
NULL	3200	3200

8 rows in set

从查询结果可以看到，COALESCE(pension,salary)函数返回的结果中，如果字段pension的值不为NULL，则该函数返回的就是COALESCE函数中第一个参数pension的值；如果字段pension的值为NULL，则该函数返回的就是COALESCE函数中第二个参数salary的值。

## 10.7 分支函数与条件表达式

在程序设计语言中，可以通过IF语句对程序进行分支控制，通过判断条件表达式是TRUE或者FALSE来决定程序的执行。在数据库中，也提供了可以实现分支控制的函数和条件表达式。不同的数据库会支持不同的分支函数。这一节就来介绍数据库中常用的分支函数与条件表达式。

### 10.7.1 IF函数

在MySQL数据库中，如果希望实现类似于编程语言中IF...ELSE...THEN的功能，可以使用IF函数实现。该函数的语法规则如下：

```
IF(expression,result1,result2)
```

该函数的功能是根据条件表达式expression判断的结果是TRUE还是FALSE，决定返回值result1还是result2。如果表达式expression的值为TRUE，则返回result1对应的结果值；如果表达式expression的值为FALSE，则返回result2对应的结果值。

IF函数也可以嵌套使用。嵌套使用的语法规则如下：

```
IF(expression,result1, IF(expression,result1,result2))
```

这里给出了两个IF函数进行嵌套的例子。当然，在IF函数中还可以允许许多层次的嵌套。下面来看一个使用嵌套IF函数实现分支控制的例子。

例10.106 根据不同的院校为教师增加工资（使用IF函数）。



## 零基础学SQL

```
SELECT teaID,teaName,dept,salary,
IF(dept='计算机系',salary+300,
IF(dept='数学系',salary+200,
IF(dept='物理系',salary+100,salary)
))AS newSalary
FROM T_teacher
```

这段SQL语句是在教师信息表（T\_teacher）中，根据教师所在的不同院系，为教师增加不同数额的工资。这里使用的是三个IF嵌套的函数实现增加不同院系教师工资的操作。第一个IF函数首先来判断字段dept表示的是否为计算机系，如果是计算机系，就将计算机系中对应的教师的工资增加300，如果字段dept表示的不是计算机系，就嵌套使用第二个IF函数判断字段dept表示的是否为数学系，如果是数学系，就将数学系中对应的教师的工资增加200，如果字段dept表示的不是数学系，就嵌套使用第三个IF函数判断字段dept表示的是否为物理系，如果是物理系，就将物理系中对应的教师的工资增加100，如果字段dept表示的不是物理系，就保持教师原来的工资不变。其查询结果如下所示。

teaID	teaName	dept	salary	newSalary
t102225	赵伟	计算机系	3000	3300
t103265	张昌	计算机系	3800	4100
t105320	于波	计算机系	2800	3100
t106358	毛翠	计算机系	4000	4300
t156354	王新	数学系	2500	2700
t156355	李中	数学系	4200	4400
t181585	李慧	物理系	3500	3600
t186585	孙立	物理系	3200	3300

8 rows in set

从查询的结果可以看出，显示结果中的最后一个字段newSalary表示教师的新工资。在教师新工资的新Salary字段中，计算机系的每一个教师的工资都加了300元，数学系的每一个教师的工资都加了200元，物理系的每一个教师的工资都加了100元。

**注意** Oracle数据库和Microsoft SQL Server数据库并不支持IF函数。在Oracle数据库中可以使用DECODE函数或者CASE表达式实现条件分支控制，在Microsoft SQL Server数据库中可以使用CASE表达式实现条件分支控制。

### 10.7.2 DECODE函数

在Oracle数据库中，如果希望实现类似于编程语言中IF...ELSE...THEN的功能，可以使用DECODE函数。该函数的语法规则如下：

```
DECODE(expression,search1,result1[,search2,result2]...[default_result])
```

该函数的功能是匹配特定表达式的结果，并将该匹配的结果返回。其中，参数expression表示指定的表达式；参数search1和参数search2表示与表达式匹配的内容；参数result1和参数result2表示要返回的结果值；参数default\_result表示要返回的默认值。其中，参数search2、result2以及参数default\_result

是可选的。

如果表达式expression的值与search1对应的值相匹配，则该函数会返回result1对应的结果值；如果表达式expression的值与search2对应的值相匹配，则该函数会返回result2对应的结果值。以此类推，如果没有任何值与表达式expression的值相匹配，则该函数返回default\_result对应的结果值。如果没有定义default\_result值，则返回NULL。

**例10.107** 根据不同的院校为教师增加工资（使用DECODE函数）。

```
SELECT teaID,teaName,dept,salary,
DECODE (dept, '计算机系', salary+300, '数学系', salary+200, '物理系', salary+100) AS newSalary
FROM T_teacher
```

这段SQL语句是在教师信息表（T\_teacher）中，根据教师所在的不同院系，为教师增加不同数额的工资。可以看到使用DECODE函数增加教师工资要比使用IF函数增加教师工资的用法简单得多。也就是说，DECODE函数要比IF函数更加灵活。

这里使用DECODE函数对表示教师所在院系的字段dept进行判断，如果字段dept表示的是计算机系，则为计算机系中的每一个教师增加300元的工资；如果字段dept表示的是数学系，则为数学系中的每一个教师增加200元的工资，如果字段dept表示的是物理系，则为物理系中的每一个教师增加100元的工资。其查询结果如下所示。

teaID	teaName	dept	salary	newSalary
t102225	赵伟	计算机系	3000	3300
t103265	张昌	计算机系	3800	4100
t105320	于波	计算机系	2800	3100
t106358	毛翠	计算机系	4000	4300
t156354	王新	数学系	2500	2700
t156355	李中	数学系	4200	4400
t181585	李慧	物理系	3500	3600
t186585	孙立	物理系	3200	3300

从查询的结果可以看出，显示结果中的最后一个字段newSalary表示教师的新工资。在该示例中，教师新工资的newSalary字段中，计算机系的每一个教师的工资都加了300元，数学系的每一个教师的工资都加了200元，物理系的每一个教师的工资都加了100元。其查询结果与例10.106相同。

**注意** MySQL数据库和Microsoft SQL Server数据库并不支持DECODE函数。在Microsoft SQL Server数据库中可以使用CASE表达式实现条件分支控制。

### 10.7.3 CASE条件表达式

在Oracle数据库（Oracle 9i及其以后的版本）、Microsoft SQL Server数据库以及MySQL数据库中，还可以使用CASE条件表达式实现分支查询的功能。该表达式可以实现多重条件分支查询的功能。CASE条件表达式有两种实现方法，一种是通过单一的条件表达式进行等值比较，一种是多种条件比较。下面就来介绍这两种CASE条件表达式的用法。

#### 1. 通过单一的条件表达式进行等值比较

在CASE条件表达式中，通过单一的条件表达式进行等值比较的语法规则如下：

## 零基础学SQL

```
CASE expression
WHEN value1 THEN result1
[WHEN value2 THEN result2
...]
[ELSE default_result]]
END
```

该条件表达式的功能是根据与expression匹配的内容，返回与之匹配的结果。其中参数expression表示指定的表达式；参数value1和参数value2表示与表达式匹配的内容；参数result1和参数result2表示要返回的结果值；参数default\_result表示要返回的默认值。其中，WHEN value2 THEN result2 以及ELSE default\_result是可选的。

如果参数expression的值与value1的值相匹配，则该函数会返回result1对应的结果值；如果表达式expression的值与value2对应的值相匹配，则该函数会返回result2对应的结果值。以此类推，如果没有任何值与表达式expression的值相匹配，则该函数返回default\_result对应的结果值。如果没有定义default\_result值，则返回NULL。

**例10.108** 根据不同的院校编号为教师增加工资。

```
SELECT teaID,teaName,dept, deptID,salary,CASE deptID
WHEN 10 THEN salary+300
WHEN 15 THEN salary+200
WHEN 18 THEN salary+100
ELSE salary END newSalary
FROM T_teacher
```

这段SQL语句是在教师信息表（T\_teacher）中，根据教师所在的不同院系编号，为教师增加不同数额的工资。这里使用CASE条件表达式对表示教师所在院系的字段deptID进行判断，如果字段deptID的值为10（即表示的是计算机系），则为计算机系中的每一个教师增加300元的工资；如果字段deptID的值为15（即表示的是数学系），则为数学系中的每一个教师增加200元的工资，如果字段deptID的值是18（即表示的是物理系），则为物理系中的每一个教师增加100元的工资。其查询结果如下所示。

teaID	teaName	dept	deptID	salary	newSalary
t102225	赵伟	计算机系	10	3000	3300
t103265	张昌	计算机系	10	3800	4100
t105320	于波	计算机系	10	2800	3100
t106358	毛翠	计算机系	10	4000	4300
t156354	王新	数学系	15	2500	2700
t156355	李中	数学系	15	4200	4400
t181585	李慧	物理系	18	3500	3600
t186585	孙立	物理系	18	3200	3300

从查询的结果可以看出，显示结果中的最后一个字段newSalary表示教师的新工资。在教师新工资的newSalary字段中，计算机系的每一个教师的工资都加了300元，可以看到计算机系中对应的deptID的值为10；数学系的每一个教师的工资都加了200元，可以看到数学系中对应的deptID的值为15；物理系的每一个教师的工资都加了100元，可以看到物理系中对应的deptID的值为18。

## 2. 多种条件比较

有些时候，需要在CASE条件表达式中对多个条件进行判断，根据不同的条件返回不同的结果，这个时候就需要使用CASE条件表达式中提供的多种比较的功能。其语法规则如下：

```
CASE
WHEN condition1 THEN result1
WHEN condition2 THEN result2
...
[ELSE default_result]]
END
```

该条件表达式的功能是返回与条件表达式匹配的结果。如果条件表达式condition1为真，则该函数会返回result1对应的结果值；如果条件表达式condition2为真，则该函数会返回result2对应的结果值。以此类推，如果没有任何条件表达式为真，则该函数返回default\_result对应的结果值。如果没有定义default\_result值，则返回NULL。

例10.109 根据教师的年龄增加相应的工资。

```
SELECT teaID,teaName,age,salary,CASE
WHEN age<=30 THEN salary+300
WHEN age<=40 THEN salary+200
WHEN age<=50 THEN salary+100
ELSE salary
END newSalary
FROM T_teacher
```

这段SQL语句中，是根据教师的年龄决定为不同教师增加不同的工资。如果教师的年龄小于30，则将教师的工资增加300；如果教师的年龄在30岁到40岁之间，则将教师的工资增加200；如果教师的年龄在40岁到50岁之间，则将教师的工资增加100；如果教师的年龄在50岁以上，则不增加教师的工资。其查询结果如下所示。

teaID	teaName	age	salary	newSalary
t102225	赵伟	38	3000	3200
t103265	张昌	43	3800	3900
t105320	于波	28	2800	3100
t106358	毛翠	50	4000	4100
t156354	王新	33	2500	2700
t156355	李中	55	4200	4200
t181585	李慧	40	3500	3700
t186585	孙立	48	3200	3300

8 rows in set

从查询的结果可以看出，显示结果中的最后一个字段newSalary表示教师的新工资。在教师新工资的newSalary字段中，年龄小于30岁的教师其工资都加了300元，年龄在30岁到40岁之间的教师其工资都加了200元，年龄在40岁到50岁之间的教师其工资都加了100元，年龄50岁以上的教师工资没有变化。

## 10.8 小结

本章主要以Oracle数据库、MySQL数据库以及Microsoft SQL Server数据库为例介绍了数据库中内置的一些常用函数。这些函数包括字符函数、数字函数、日期时间函数、转换函数、比较函数、空值处理函数以及分支函数和条件表达式。在使用函数查询数据时，数据库中的数据本身并没有发生改变。

希望通过这一章的学习，读者可以了解这些常用函数的功能以及它们的使用方法。这里介绍的是一些在实际应用中可能会经常用到的函数，在数据库中还有一些类似于这一章讲的内置函数没有介绍。由于篇幅所限，这里不可能将数据库中所有的内置函数都一一讲到。





## 第11章 视图的创建与删除

视图可以认为是从一个数据表（或者视图）或者是多个数据表（或者视图）中导出的表。视图本身没有任何数据，它只是用来存放视图的定义，因此视图只是虚拟的表。如果数据表中的数据记录发生变化，与该数据表有关的视图中的数据记录也会随之变化。这一章主要介绍视图的作用以及如何创建和删除一个视图。

本章重点：

- 视图的作用
- 视图的创建
- 视图的删除

### 11.1 视图的作用

在实际应用中，创建一个视图主要基于两个方面的考虑。一方面视图可以提高数据访问的安全性，另一方面视图可以为复杂查询操作带来方便。

#### 1. 提高数据访问的安全性

在实际开发过程中，有些时候并不希望开发人员或者用户对数据表中的所有记录都可以进行查询操作。例如，在教师信息表中，教师的姓名、所在院系等信息是可以让所有的用户都知道的，而教师的工资以及津贴等信息并不希望所有的用户都知道。这个时候可以创建一个视图，将不希望所有用户看到的信息隐藏起来，用户查询视图时，有关教师的工资以及津贴等信息是查询不到的。

#### 2. 方便查询操作

有些时候，数据表之间的某些数据会经常用到，如果在不同的地方要经常查询这些数据记录，就需要多次使用SELECT语句重复查询这些信息，有些时候这些SELECT语句可能会很复杂，显然这种做法给程序的编写和用户对数据的访问带来了不便。这时可以创建一个视图，将在数据表中经常用到的数据信息都放到这个视图中。这样每次开发人员或者用户在使用和查询这些数据时，就可以直接通过视图查询，而不必再写复杂的SELECT语句了。这种情况对于经常需要通过多表连接进行复杂查询操作更为适用。

通过创建视图可以提高数据访问的安全性，另外创建视图还可以简化对复杂查询的操作，方便了数据的查询。

### 11.2 创建视图

创建视图可以使用CREATE VIEW语句。可以基于单表创建一个视图，也可以基于多表连接创建一

个视图，基于函数、分组数据创建视图，还可以基于一个已有视图创建新的视图，另外通过在CREATE VIEW语句中添加WITH CHECK OPTION、WITH READ ONLY等不同的关键字，还可以限制视图的更新操作。这一节就来介绍如何使用CREATE VIEW语句创建不同的视图。

### 11.2.1 基于单表创建视图

基于单表创建的视图是最简单的视图，在这个视图中只是将一个数据表中经常用到的一些数据列提取出来，创建的视图并不包含函数、表达式、分组函数等内容。使用CREATE VIEW语句创建简单视图的语法格式如下：

```
CREATE VIEW view_name([column_name1[,column_name2]...]  
AS subquery
```

其中，CREATE VIEW是关键字，表示要创建一个视图；view\_name用来指定视图的名字；column\_name1、column\_name2表示创建视图中列的名字，这里可以指定一列，也可以指定多列，还可以不指定。如果不指定列的名字，则视图中列的名字与子查询中数据表中查询的列的名字相同；subquery表示创建视图对应的子查询语句。

例11.1 为教师信息表创建一个视图。

```
CREATE VIEW V_teacher  
AS  
SELECT teaID,teaName,age,sex,dept,profession  
FROM T_teacher
```

这段SQL语句是基于教师信息表子查询中指定的列创建了一个视图。其中，V\_teacher指定了视图的名字；关键字AS的后面跟的是一个子查询语句。在这个子查询语句中，是从教师信息表T\_teacher查询出指定的列，并将该表中查询出来的数据列作为视图V\_teacher中的列，视图V\_teacher中列的名字与子查询里SELECT语句中指定的列的名字相同。

选中MySQL 5.0用户图形界面的右侧Schemata选项下test\_STInfo数据库，单击鼠标右键，选择“Refresh”选项（或者按F5键，刷新test\_STInfo数据库）。刷新操作执行后，会在test\_STInfo数据库下方看到一个名为V\_teacher的视图，如图11.1所示。

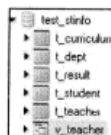


图11.1 创建V\_teacher视图

视图创建完成后，可以使用SELECT语句对视图中的信息执行查询操作。为了了解视图中都有哪些列，这里在SELECT语句中使用“\*”号查询视图中的所有列。其SQL语句如下：

```
SELECT *  
FROM V_teacher
```

这段SQL语句是查询视图V\_teacher中所有列的信息。这里在SELECT语句中使用“\*”号查询视图中的所有列。其查询结果如图11.2所示。

teaID	teaName	age	sex	dept	profession
t102225	赵伟	38	男	计算机系	副教授
t103285	张磊	43	男	计算机系	教授
t105320	于波	28	男	计算机系	讲师
t106358	毛翠	51	女	计算机系	教授
t156354	王新	33	女	数学系	讲师
t156365	李中	55	女	数学系	教授
t181585	李慧	40	女	物理系	教授
t185585	孙立	48	男	物理系	讲师

图11.2 查询视图V\_teacher

从图11.2中可以看到，视图中一共包含了6列。分别是列teaID、列teaName、列age、列sex、列dept和列profession。它们分别用来表示教师编号、教师姓名、教师年龄、教师性别、教师所在院系和教师职称信息。这6个列的名字就是在例11.1中子查询语句中SELECT语句指定列的名字。

视图V\_teacher中并不存在教师工资、教师津贴这样的敏感信息，而只是包含了教师的一些基本信

息。这样，以后再需要查询教师基本信息时，只要查询视图V\_teacher就可以了。通过查询视图V\_teacher也保证了与教师工资和教师津贴有关的敏感信息不会被泄露。

当然，也可以不使用子查询中指定的列的名字，而是为创建的视图的列起其他的名字。这个时候就需要在创建视图的CREATE VIEW语句中，在视图名的后面指定视图中列的名字。

**例11.2** 为教师信息表创建一个视图，并为视图创建别名。

```
CREATE VIEW V_teacher2(v_teaID,v_teaName,v_age,v_sex,v_dept,v_profession)
AS
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
```

这段SQL语句也是基于教师信息表子查询中指定的列创建了一个视图，只不过为视图V\_teacher2指定了列的名字。这里在视图名V\_teacher2的后面指定了视图中列的别名。此时使用SELECT\*语句查询视图V\_teacher2，其查询结果如图11.3所示。

v_teaID	v_teaName	v_age	v_sex	v_dept	v_profession
t102225	赵伟	38	男	计算机系	副教授
t103255	张磊	43	男	计算机系	教授
t105320	于波	28	男	计算机系	讲师
t106358	毛翠	51	女	计算机系	教授
t156354	王新	33	女	数学系	讲师
t156395	李中	55	女	数学系	教授
t181585	李慧	40	女	物理系	教授
t186585	孙立	48	男	物理系	讲师

图11.3 查询视图V\_teacher2

从图11.3中可以看到，视图中一共包含了6列。分别是列v\_teaID、列v\_teaName、列v\_age、列v\_sex、列v\_dept和列v\_profession。它们分别用来表示教师编号、教师姓名、教师年龄、教师性别、教师所在院系和教师职称信息。这6个列的名字就是在例11.2中视图名V\_teacher2后面的括号中指定列的名字。

#### 注意

首先，如果创建视图的子查询语句包含有表达式或函数，则在创建视图时需要为视图定义列名。其次，如果要在视图名的后面指定列的名字，那么其指定列的个数要与子查询语句里从数据表中查询出来的列的个数相同。最后，如果视图中指定列的个数与子查询语句里从数据表中查询出来的列的个数不同，则创建视图的语句在执行时会出现错误。

### 11.2.2 基于多表连接创建视图

基于多表连接创建的视图，是指将多个数据表中经常用到的数据列使用WHERE子句连接起来组成的视图。创建基于多表连接的视图，主要是为了简化查询语句，提高查询效率。用CREATE VIEW语句创建连接视图的语法格式如下：

```
CREATE VIEW view_name([column_name1[,column_name2]...])
AS subquery
```

其中，CREATE VIEW是关键字，表示要创建一个视图；view\_name用来指定视图的名字；column\_name1、column\_name2表示创建视图中列的名字，这里可以指定一列，也可以指定多列，还可以不指定。如果不指定列的名字，则视图中列的名字与子查询中数据表中查询的列的名字相同；subquery表示创建视图对应的子查询语句。

这里与11.2.1中创建简单视图的不同就在于子查询。基于单表创建的视图的子查询只是对其中一个数据表进行查询，而基于多表连接创建视图中的子查询中是需要将多个数据表使用WHERE连接起来的。

**例11.3** 创建学生成绩视图，要求视图中的学生的成绩都要在及格以上。

```
CREATE VIEW v_result
AS
```

```
SELECT R.stuID,C.curID, C.curName,R.result
FROM T_result R,T_curriculum C
WHERE R.curID=C.curID
AND R.result>=60
```

这段SQL语句是基于课程信息表和成绩信息表两个表连接创建一个连接视图。其中，V\_result指定了视图的名字；关键字AS的后面跟的是一个子查询语句。在这个子查询语句中，T\_result表示成绩信息表，T\_curriculum表示课程信息表，WHERE子句中，R.curID=C.curID用来指定两个表的连接条件，其中R和C表示成绩信息表T\_result和课程信息表T\_curriculum的别名，AND子句之后的R.result>=60用来限定查询条件，这里将学生的课程成绩限制在60分以上。

选中MySQL 5.0用户图形界面的右侧Schemata选项下test\_STInfo数据库，单击鼠标右键，选择“Refresh”选项（或者按F5键，刷新test\_STInfo数据库）。刷新操作执行后，会在test\_STInfo数据库下方看到一个名为V\_result的视图。

视图创建完成后，可以使用SELECT语句对视图中的信息执行查询操作。为了了解视图中都有哪些列，这里在SELECT语句中使用“\*”号查询视图中的所有列。其SQL语句如下：

```
SELECT *
FROM V_result
```

这段SQL语句是查询视图V\_result中所有列的信息。这里在SELECT语句中使用“\*”号查询视图中的所有列。其查询结果如图11.4所示。

从图11.4中可以看到，视图中一共包含了4列。分别是列stuID、列curID、列curName、列result，它们分别用来表示学生编号、课程编号、课程名称和课程成绩。这4个列的名字就是在例11.3中子查询语句中SELECT语句指定列的名字。

stuID	curID	curName	result
s102203	i105	计算机系统结构	85
s206363	i105	计算机系统结构	80
s102203	i232	数据库基础	75
s221296	i232	数据库基础	60
s2532653	i232	数据库基础	70
s102203	i321	C语言	90
s2532653	i321	C语言	90
s102203	i333	高等数学	60

图11.4 查询视图V\_result

### 11.2.3 基于函数、分组数据创建视图

基于函数、表达式创建视图，就是指在子查询语句中包含有函数、表达式以及对数据表中的数据分组排序的视图。基于函数、分组数据的视图主要是为了简化查询语句，提高查询效率。用CREATE VIEW语句创建连接视图的语法格式如下：

```
CREATE VIEW view_name([column_name1[,column_name2]...]
AS subquery
```

其中，CREATE VIEW是关键字，表示要创建一个视图；view\_name用来指定视图的名字；column\_name1、column\_name2表示创建视图中列的名字，这里可以指定一列，也可以指定多列，还可以不指定。如果不指定列的名字，则视图中列的名字与子查询中数据表中查询的列的名字相同；subquery表示创建视图对应的子查询语句。

这里与11.2.1中创建简单视图的不同就在于子查询。基于函数、分组数据创建视图的子查询中可以包含有函数、表达式以及对数据表中的数据进行分组排序等操作。

例11.4 为教师信息表创建视图，要求对院校和教师职称进行分组并且所有教师的工资要大于3000。

```
CREATE VIEW v_teacher_salary
AS
```

```
SELECT dept,profession,MAX(salary) AS maxsalary
FROM T_teacher
GROUP BY dept,profession
HAVING MAX(salary)>3000
```

这段SQL语句是为教师信息表创建视图，该视图中所有的教师工资都大于3000。其中，V\_teacher\_salary指定了视图的名字；关键字AS的后面跟的是一个子查询语句。在这个子查询语句中，T\_teacher表示教师信息表，GROUP BY指定了分组条件，这里以教师所在院系和教师职称进行分组。其中列dept表示教师所在院系，列profession表示教师职称。

选中MySQL 5.0用户图形界面的右侧Schemata选项下test\_STInfo数据库，单击鼠标右键，选择“Refresh”选项（或者按F5键，刷新test\_STInfo数据库）。刷新操作执行后，会在test\_STInfo数据库下方看到一个名为V\_teacher\_salary的视图。

视图创建完成后，可以使用SELECT语句对视图中的信息执行查询操作。为了了解视图中都有哪些列，这里在SELECT语句中使用“\*”号查询视图中的所有列。其SQL语句如下：

```
SELECT *
FROM v_teacher_salary
```

这段SQL语句是查询视图V\_teacher\_salary中所有列的信息。这里在SELECT语句中使用“\*”号查询视图中的所有列。其查询结果如图11.5所示。

dept	profession	maxsalary
计算机系	教授	4000
数学系	教授	4200
物理系	讲师	3200
物理系	教授	3500

图11.5 查询视图V\_teacher\_salary

从图11.5中可以看到，视图中一共包含了3列。分别是列dept、列profession、列maxsalary，它们分别用来表示教师所在院系、教师职称和教师工资。这3个列的名字就是在例11.4中子查询语句中SELECT语句指定列的名字。

#### 11.2.4 为视图添加CHECK约束

在创建视图时，还可以为视图添加CHECK约束。如果在创建的视图添加了CHECK约束条件，那么在视图中执行插入、修改和删除等更新语句时，其插入、修改和删除的数据必须满足视图定义中查询表达式中的查询条件。在创建视图时，为视图添加CHECK约束条件的语法格式如下：

```
CREATE VIEW view_name([column_name1[,column_name2]...])
AS subquery
WITH CHECK OPTION
```

其中，CREATE VIEW是关键字，表示要创建一个视图；view\_name用来指定视图的名字；column\_name1、column\_name2表示创建视图中列的名字，这里可以指定一列，也可以指定多列，还可以不指定。如果不指定列的名字，则视图中列的名字与子查询中数据表中查询的列的名字相同；subquery表示创建视图对应的子查询语句；WITH CHECK OPTION表示对创建的视图添加CHECK约束条件。当视图执行数据的增加、修改和删除等更新操作时，会对其做条件检查。

例11.5 为教师信息表中院系编号为t\_10的教师创建视图，并为该视图添加CHECK约束条件。

```
CREATE VIEW V_teacher_dept
AS
SELECT teaID,teaName,age,sex, deptID,dept,profession
FROM T_teacher
```



## 零基础学SQL

```
WHERE deptID='t_10'  
WITH CHECK OPTION
```

这段SQL语句是为教师信息表中院系编号为t\_10的教师创建视图，并为该视图添加CHECK约束条件。其中，V\_teacher\_dept指定了视图的名字；关键字AS的后面跟的是一个子查询语句。在这个子查询语句中，T\_teacher表示教师信息表，WHERE子句限定了查询条件，查询院系编号为t\_10的教师信息。列deptID表示教师所在院系编号；WITH CHECK OPTION表示对创建的视图添加CHECK约束条件。

选中MySQL 5.0用户图形界面的右侧Schemata选项下test\_STInfo数据库，单击鼠标右键，选择“Refresh”选项（或者按F5键，刷新test\_STInfo数据库）。刷新操作执行后，会在test\_STInfo数据库下方看到一个名为V\_teacher\_dept的视图。

视图创建完成后，可以使用SELECT语句对视图中的信息执行查询操作。为了了解视图中都有哪些列，这里在SELECT语句中使用“\*”号查询视图中的所有列。其SQL语句如下：

```
SELECT *  
FROM V_teacher_dept
```

这段SQL语句是查询视图V\_teacher\_dept中所有列的信息。这里在SELECT语句中使用“\*”号查询视图中的所有列。其查询结果如图11.6所示。

teaID	teaName	age	sex	deptID	dept	profess
t102225	赵伟	38	男	t_10	计算机系	副教授
t103265	张磊	43	男	t_10	计算机系	教授
t105320	于峻	28	男	t_10	计算机系	讲师
t106358	毛翠	51	女	t_10	计算机系	教授

图11.6 查询视图V\_teacher\_dept

从图11.6中可以看到，V\_teacher\_dept视图中，教师所在院系编号deptID为t\_10，表示的是计算机系。V\_teacher\_dept视图中显示的教师都是计算机系的教师信息。

因为在V\_teacher\_dept视图中定义了CHECK约束条件，所以如果要对V\_teacher\_dept视图进行数据插入操作时，其教师编号deptID必须指定为t\_10；如果要对V\_teacher\_dept视图进行数据更新操作时，不能对列deptID进行修改，只能修改视图中除院系编号deptID以外的其他列的信息。

### 11.2.5 基于一个已有视图创建新的视图

除了可以基于数据表创建视图之外，还可以基于一个已有的视图来创建一个新的视图。基于一个已有视图创建新的视图主要是为了简化查询语句，提高查询效率。例如，现在想要查询学生成绩最高分在80分以上的学生信息。如果不使用视图，则其查询语句如下：

```
SELECT A.stuID,A.curID, A.curName,MAX(A.result)  
FROM(  
SELECT R.stuID,C.curID, C.curName,R.result  
FROM T_result R,T_curriculum C  
WHERE R.curID=C.curID  
)A  
GROUP BY A.stuID  
HAVING MAX(A.result)>80
```

这段SQL语句中，在FROM语句中有一个子查询语句，这个子查询语句用来连接课程信息表和成绩信息表，并将两个表连接后的信息组成一个新的表，为这个表起个别名叫做A表，还使用GROUP BY子句对表示学生编号的列stuID进行分组，从分组中筛选出成绩最高分在80分以上的学生信息。可以看到，这个SQL语句是比较复杂的。其查询结果如图11.7所示。

现在如果在程序中需要经常用到成绩最高分在80分以上的学生信息，显然上面的SQL语句就比较复杂了，那么有什么办法可以简化上面的查询语句，提高查询效率呢？是有办法的，那就是使用视图。可以把上面的SQL语句通过使用CREATE VIEW语句创建一个新的视图，这个读者应该很容易创建，这里就不讲了。这里讲解创建视图的另外一种方法，使用一个已有视图创建一个新的视图。

stuID	curID	curName	MAX(result)
s102203	i105	计算机系统结构	90
s206363	i105	计算机系统结构	80
s2532653	i232	数据库基础	90

图11.7 查询成绩最高分在80分以上的学生信息

在例11.3中，使用多表连接创建了一个学生成绩视图，该视图中的学生的成绩都要在及格以上，可以基于这个视图，创建一个新的视图，用来存储成绩最高分在80分以上的学生信息。

**例11.6** 基于一个已有视图创建新的视图，要求该视图中学生成绩最高分都在80分以上。

```
CREATE VIEW V_result2
AS
SELECT stuID,curID, curName,MAX(result)
FROM V_result
GROUP BY stuID
HAVING MAX(result)>=80
```

这段SQL语句是基于一个已有视图创建新的视图，而且该视图中学生成绩最高分都在80分以上。其中，V\_result2指定了视图的名字；关键字AS的后面跟的是一个子查询语句。在这个子查询语句中，T\_result表示成绩信息表，GROUP BY指定了分组条件，这里以学生编号进行分组。其中列stuID表示学生编号；HAVING子句用来对分组后的信息进行筛选，将成绩最高分在80分以上的学生全部查询出来。

选中MySQL 5.0用户图形界面的右侧Schemata选项下test\_STInfo数据库，单击鼠标右键，选择“Refresh”选项（或者按F5键，刷新test\_STInfo数据库）。刷新操作执行后，会在test\_STInfo数据库下方看到一个名为V\_result2的视图。

以后如果希望查询成绩最高分都在80分以上的学生信息，只需要查询视图V\_result2就可以了。其SQL语句如下：

```
SELECT *
FROM V_result2
```

这段SQL语句是查询视图V\_result2中所有列的信息。这里在SELECT语句中使用“\*”号查询视图中的所有列。其查询结果如图11.8所示。

stuID	curID	curName	MAX(result)
s102203	i105	计算机系统结构	90
s206363	i105	计算机系统结构	80
s2532653	i232	数据库基础	90

图11.8 查询成绩最高分在80分以上的学生信息

从图11.8中可以看到，通过基于一个已有视图创建新的视图，视图中所有学生的成绩最高分都在80分以上。与图11.7中显示的查询结果相同。

### 11.2.6 创建只读视图

在Oracle数据库中，还可以使用WITH READ ONLY关键字创建一个只读视图，只读视图只能用于数据查询，而不能用于数据的插入、修改和删除等视图的更新操作。创建只读视图的语法格式如下：

```
CREATE VIEW view_name([column_name1[,column_name2]...])
AS subquery
WITH READ ONLY
```

## 零基础学SQL

其中，CREATE VIEW是关键字，表示要创建一个视图；view\_name用来指定视图的名字；column\_name1、column\_name2表示创建视图中列的名字，这里可以指定一列，还可以指定多列，还可以不指定。如果不指定列的名字，则视图中列的名字与子查询中数据表中查询的列的名字相同；subquery表示创建视图对应的子查询语句；WITH READ ONLY表示创建的视图是只读视图。

例11.7 为教师信息表创建视图，并将该视图定义为只读视图。

```
CREATE VIEW V_teacher3
AS
SELECT teaID,teaName,age,sex,dept,profession
FROM T_teacher
WITH READ ONLY
```

这段SQL语句与例11.1中的基于教师信息表创建视图的SQL语句基本相同，只是在创建视图的最后加了关键字WITH READ ONLY，表示这个视图只能进行查询操作，不能进行数据更新操作。

### 11.3 删除视图

如果在实际使用中某一个视图已经不再需要，可以使用DROP VIEW语句将其删除。其语法格式如下：

```
DROP VIEW view_name
```

其中，DROP VIEW是用于删除视图的关键字；view\_name表示要删除的视图的名字。一般用户如果要删除视图，需要具有删除视图的权限。（有关权限的授予的内容将在第15章介绍）

例11.8 删除V\_teacher视图。

```
DROP VIEW V_teacher
```

这段SQL语句是要将V\_teacher视图删除。其中V\_teacher表示视图的名字。

### 11.4 小结

这一章主要介绍了视图的作用以及创建和删除一个视图的方法。创建视图主要是从安全性和简化查询操作两个方面来考虑。创建一个视图既可以基于一个数据表，也可以基于多个数据表，还可以基于一个已有的视图。在创建视图的语句中添加WITH CHECK OPTION、WITH READ ONLY等不同的关键字，还可以限制视图的更新操作。

视图可以像数据表一样，进行查询和更新操作。可以使用SELECT语句对视图进行查询操作，视图的查询操作与数据表的查询操作相同。也可以使用INSERT、UPDATE、DELETE语句对视图中的数据进行更新操作。有关视图中数据的插入、修改和删除的内容将在第12章、第13章和第14章中介绍。

## 第四篇

# 数据更新

## 第12章 插入数据记录

数据操纵语言包括INSERT、UPDATE和DELETE。INSERT语句主要是用来执行数据的插入操作。使用INSERT语句既可以插入单行数据，也可以使用子查询插入多行数据；既可以向数据表中插入数据记录，也可以向视图中插入数据记录。但是在向视图插入数据记录时，还需要有一些额外的限制。这一章就主要介绍如何使用INSERT INTO语句向数据表和视图中插入数据记录。

本章重点：

- ☐ 插入单行数据记录
- ☐ 向定义有外键约束的表中插入数据记录
- ☐ 利用MySQL 5.0数据库一次插入多条数据记录
- ☐ 使用子查询插入多行数据实现表中数据的复制
- ☐ 向视图中插入数据记录

### 12.1 向数据表中插入数据记录

如果想向数据表中增加数据记录，可以使用INSERT INTO语句。使用INSERT INTO语句既可以向数据表中插入单行数据记录，也可以向数据表中插入多行数据记录，而且利用MySQL 5.0数据库的用户图形界面提供的功能还可以向指定的数据表中一次插入多条数据记录。这一节就来介绍使用INSERT INTO语句向数据表中增加数据的方法。

#### 12.1.1 插入单行数据记录

使用INSERT INTO语句可以向数据表中插入单行数据记录。使用INSERT INTO语句向数据表中插入单行数据记录的语法格式如下：

```
INSERT INTO table_name[(column_name1,column_name2,...)]  
VALUES (value1[,value2]...)
```

其中，INSERT INTO表示向数据表插入数据记录的关键字；table\_name表示表的名字；column\_name1、column\_name2用来指定表中列的名字，多个列名之间需要使用逗号分隔。它们是可选的；关

键字VALUES后面括号中的value1、value2表示向数据表中插入的数据。

注意

如果INSERT INTO语句中在表的名字后面没有指定列的名字，那么关键字VALUES后面括号中的列必须要与表中原有列的顺序相对应，其数据类型要与表中原有列的数据类型一致，value值的个数要与表中原有列的个数相同；如果INSERT INTO语句中在表的名字后面有指定列的名字，那么关键字VALUES后面括号中的列要与INSERT INTO语句中表名后面括号中指定列的顺序相对应，其数据类型要与INSERT INTO语句中表名后面括号中指定列的数据类型一致，value值的个数要与表名后面括号中指定列的个数相同。

例12.1 向学生信息表中插入一条学生记录。

```
INSERT INTO T_student
VALUES
('s281234','王龙',20,'男','19890218')
```

这段SQL语句是向学生信息表中插入了一条学生记录。其中，T\_student表示学生信息表；关键字VALUES后面的括号中指定了向学生信息表中插入的值。值“s281234”表示学生编号，值“王龙”表示学生的姓名，值“20”表示学生的年龄，值“男”表示学生的性别，值“19890218”表示学生的出生日期。

这里插入的值的顺序与学生信息表T\_student中原有列的顺序一致，其数据类型要与表中原有列的数据类型一致，并且插入值的个数也要与表中原有列的个数相同。

注意

首先，如果向数据表插入的值是数字，则可以直接使用数字值；如果向数据表中插入的值是字符或者是日期类型的数据，则需要使用单引号将其引住。

其次，在向数据表插入数据时，需要满足数据表中的约束条件，必须向数据表中定义为主键约束的列和定义为非空约束的列插入数据记录。如果没有向主键列和非空约束的列插入数据记录，则数据库管理系统会报错，拒绝执行插入操作。

最后，在使用INSERT INTO语句插入数据时，如果某个列允许插入空值，要想向该列插入空值，则需要插入NULL代替；如果该列定义了NOT NULL约束，即不允许插入空值，那么就不能向该列插入NULL，否则数据库管理系统会报错，但是可以向该列插入一个空格代替。

插入数据的SQL语句正确执行后，可以使用SELECT语句查询教师信息表中的学生信息。其SQL语句如下：

```
SELECT stuID,stuName,age,sex,birth
FROM T_teacher
```

这段SQL语句是查询学生信息表中的学生信息。其查询结果如图12.1所示。

从图12.1可以看到，在学生信息表中确实增加了一条学生编号为s281234，学生姓名为王龙的学生记录。

当然，也可以在INSERT INTO语句中在表名后面指定列的名字。如果在INSERT INTO语句中在表名后面指定列的名字，那么关键字VALUES后面括号中的value值就需要与INSERT INTO语句中表名后面括号中指定列的顺序相对应，value值的个数要与表名后面括号中指定列的个数相同。

stuID	stuName	age	sex	birth
s102203	赵亮	23	男	1986-05-16 00:00:00
s112303	郑磊	21	女	1988-01-25 00:00:00
s115263	王海	23	男	1986-08-02 00:00:00
s206363	张明	22	男	1987-04-23 00:00:00
s221256	王昌鹤	24	男	1985-03-18 00:00:00
s231456	王玉梅	22	女	1987-03-28 00:00:00
s23256	李玉峰	24	女	1985-06-16 00:00:00
s2532653	李凤	24	女	1985-06-06 00:00:00
s281234	王龙	20	男	1989-02-18 00:00:00

图12.1 查询学生信息表



### 例12.2 向学生信息表中插入一条学生记录（在INSERT INTO语句中使用列名）。

```
INSERT INTO T_student(stuID,stuName,age,sex,birth)
VALUES
('s284321','李茜',20,'女','19890820')
```

这段SQL语句是向学生信息表中插入了一条学生记录，并在INSERT INTO语句表名的后面指定了列的名字。其中T\_student表示学生信息表；关键字VALUES后者的括号中指定了向学生信息表中插入的值。值“s284321”对应数据表中的列stuID，值“李茜”对应数据表中的列stuName，值“20”对应数据表中的列age，值“女”对应数据表中的列sex，值“19890820”对应数据表中的列birth。这里插入的值与表名T\_student后指定的值都是一一对应的。

### 12.1.2 向定义有外键约束的表中插入数据记录

向定义有外键约束的表中插入数据记录时，其插入的数据记录需要满足外键约束条件。例如，对于成绩信息表T\_result，该表中定义了一个指向学生信息表的外键约束，其删除方式（ON DELETE）和修改方式（ON UPDATE）都是RESTRICT。如果向成绩信息表T\_result中插入一条在学生信息表T\_student中不存在的数据记录，则数据库管理系统会报错，拒绝执行插入操作。

### 例12.3 向成绩信息表中插入一条在学生信息表中不存在的数据记录。

```
INSERT INTO T_result
VALUES
('s111111','t321',75)
```

这段SQL语句是向成绩信息表T\_result中插入一条学生编号为s111111的学生成绩记录。由于在学生信息表中不存在学生编号为s111111的学生信息，因此在向成绩信息表中执行插入操作时，MySQL数据库管理系统会显示如下的错误信息。

```
Cannot add or update a child row: a foreign key constraint fails (`test_stinfo/t_result`,
CONSTRAINT `t_result_ibfk_1` FOREIGN KEY (`stuID`) REFERENCES `t_student` (`stuID`))
```

这个错误信息表示数据表之间定义了外键约束关系，向成绩信息表T\_result中增加学生编号为s111111的成绩记录违反了外键约束条件。在成绩信息表T\_result中已经定义了一个指向学生信息表T\_student的外键约束，由于学生信息表中没有学生编号为s111111的学生记录，因此不能向成绩信息表中插入学生编号为s111111的成绩记录。

所以在向定义有外键约束的表中插入数据记录时，其插入的数据记录需要满足外键约束条件。如果希望向成绩信息表T\_result中插入一条学生的成绩记录，那么在学生信息表中应该存在该学生的信息。

### 例12.4 向成绩信息表中插入一条学生的成绩记录，该学生信息存在于学生信息表中。

```
INSERT INTO T_result
VALUES
('s281234','t321',75);
```

这段SQL语句是向成绩信息表T\_result中插入一条学生编号为s281234的学生成绩记录。学生编号为s281234的学生记录是在学生信息表中已经存在的记录。因此例12.4中的这条插入语句是可以执行的。

插入数据的SQL语句正确执行后，可以使用SELECT语句查询成绩信息表中学生编号为s281234的学生成绩信息。其SQL语句如下：

```
SELECT stuID, curID, result
FROM T_result
WHERE stuID = 's281234'
```

这段SQL语句是查询成绩信息表中的学生编号为s281234的学生成绩信息。其查询结果如图12.2所示。

stuID	curID	result
s281234	t321	75

图12.2 查询成绩信息表学生编号为s281234的学生成绩信息

从图12.2可以看到，在成绩信息表中确实存在一条学生编号为s281234，课程编号为t321的成绩记录。

**注意** 在向定义有外键约束的表中插入数据记录时，其插入的数据记录需要满足外键约束条件。

### 12.1.3 使用子查询插入多行数据实现表中数据的复制

使用INSERT INTO语句可以向数据表中插入多行数据记录。使用INSERT INTO语句向数据表中插入多行数据记录的语法格式如下：

```
INSERT INTO table_name[(column_name1,column_name2...)]
subquery
```

其中，INSERT INTO表示向数据表插入数据记录的关键字；table\_name表示表的名字；column\_name1、column\_name2用来指定表中列的名字，多个列名之间需要使用逗号分隔。它们是可选的；subquery用来表示向数据表中插入数据的子查询语句。

**注意**

在INSERT INTO语句中使用子查询向数据表中插入数据时，如果INSERT INTO语句中在表的名字后面没有指定列的名字，那么子查询语句中的列必须要与表中原有列的顺序相对应，其数据类型要与表中原有列的数据类型一致，value值的个数要与表中原有列的个数相同；如果INSERT INTO语句中在表的名字后面有指定列的名字，那么子查询语句中的列必须要与表中指定列的顺序相对应，其数据类型要与表中指定列的数据类型一致，value值的个数要与表中指定列的个数相同。

在INSERT INTO语句中使用子查询插入多行数据，主要的功能是可以实现数据表中数据的复制。在9.7节中曾经介绍过一种实现数据表中数据复制的方法，那是通过在CREATE TABLE语句中使用子查询来建立一张新的数据表，同时将原有表中的数据插入到新建的数据表中，实现数据表中数据的复制功能。这里介绍的方法是通过在INSERT INTO语句中使用子查询插入多行数据的方法实现数据表中数据的复制。

这里以教师信息表T\_teacher为例，在INSERT INTO语句中使用子查询插入多行数据的方法复制教师信息表T\_teacher中的数据。

(1) 为了在INSERT INTO语句中使用子查询的方法实现数据表中数据的复制，首先需要创建一个新的数据表，这个新的数据表的表结构与教师信息表T\_teacher的表结构相同。其创建新的教师信息表的SQL语句如下：

```
CREATE TABLE T2_teacher (
teaID VARCHAR (15) PRIMARY KEY,
teaName VARCHAR (10) NOT NULL,
age INT NOT NULL,
```

```
sex VARCHAR (2) NOT NULL,  
deptID VARCHAR (15),  
dept VARCHAR (20) NOT NULL,  
profession VARCHAR (10) ,  
salary INT NOT NULL,  
pension DOUBLE  
)
```

这段SQL语句是创建一个新的教师信息表T2\_teacher，在T2\_teacher表中一共包括了9个字段，其中，teaID表示教师编号，teaName表示教师姓名，age表示教师年龄，sex表示教师性别，deptID表示教师所在院系编号，dept表示教师所在院系，profession表示教师职称，salary表示教师工资，pension表示教师津贴。列teaID作为该表的主键。它与教师信息表T\_teacher的表结构完全相同。

(2) 在INSERT INTO语句中使用子查询，将教师信息表T\_teacher中的数据全部复制到新的教师信息表T2\_teacher中。其SQL语句如下：

```
INSERT INTO T2_teacher  
SELECT teaID,teaName,age,sex,deptID,dept,profession,salary,pension  
FROM T_teacher
```

这段SQL语句是将教师信息表T\_teacher中的教师记录全部复制给新的教师信息表T2\_teacher。其中，T2\_teacher表示新创建的教师信息表；子查询语句是查询原有教师信息表T\_teacher中的教师记录。

(3) 正确执行完上述两个SQL语句之后，在MySQL 5.0的用户图形界面中的下方会看到如下的提示信息：“8 rows affected by the last command,no resultset returned”表明教师信息表T\_teacher中的8条教师记录已经全部复制到新的教师信息表T2\_teacher中了。

(4) 为了确定教师信息表T\_teacher中的教师记录是否已经全部复制到新的教师信息表T2\_teacher中了，这里使用SELECT语句对新的教师信息表T2\_teacher进行查询。其查询的SQL语句如下：

```
SELECT teaID,teaName,age,sex,deptID,dept,profession,salary,pension  
FROM T2_teacher
```

这段SQL语句是查询新的教师信息表T2\_teacher中的教师记录。其查询结果如图12.3所示。

teaID	teaName	age	sex	deptID	dept	profession	salary	pension
t102225	赵伟	38	男	L10	计算机系	副教授	3000	260.5
t103265	张晶	43	男	L10	计算机系	教授	3800	300
t105320	于波	28	男	L10	计算机系	讲师	2900	228
t106358	毛翠	51	女	L18	计算机系	教授	4000	289.3
t156354	王新	33	女	L15	数学系	讲师	2500	153.8
t156355	李中	55	女	L15	数学系	教授	4200	310.2
t181585	李慧	40	女	L18	物理系	教授	3500	278
t186585	孙立	48	男	L18	物理系	讲师	3200	253.8

图12.3 查询新的教师信息表T2\_teacher中的教师记录

#### 12.1.4 利用MySQL 5.0数据库一次插入多条数据记录

在MySQL 5.0的用户图形界面中，提供了可以一次执行多条SQL语句的功能。通过MySQL 5.0用户图形界面提供的这个功能，可以将多条SQL语句放到一个脚本去中一起执行。下面就来介绍一下使用MySQL 5.0用户图形界面一次执行多条插入的SQL语句的使用方法。

(1) 选择“开始”|“所有程序”|“MySQL”菜单，选中“MySQL Query Brower”选项，输入用户名、密码，在Default Schema选项对应的文本框中输入一个数据库的名字test\_STInfo，进入到MySQL

零基础学SQL

5.0的用户图形界面中。

- (2) 选择“File”菜单，在对应的下拉框中选择“New Script Tab”选项，如图12.4所示。
- (3) 选中“New Script Tab”选项后，在MySQL 5.0的用户图形界面会打开一个输入SQL语句的脚本区。在该脚本区中可以输入多条SQL语句。如图12.5所示。

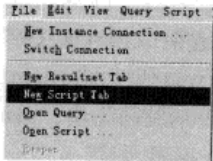


图12.4 选择“New Script Tab”选项

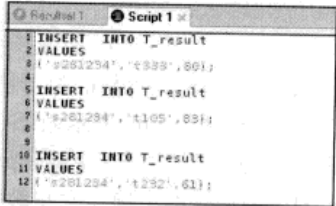


图12.5 SQL语句的脚本区

在图12.5中，一共输入了3条SQL语句，这3条SQL语句是在成绩信息表中为学生编号为s281234的学生插入课程的成绩信息。这里的每一个SQL语句之间需要用分号（；）将其分开。

在MySQL 5.0的用户图形界面的工具栏中，有一个“Execute”按钮。单击“Execute”按钮可以将脚本区中输入的SQL语句一次全部执行。

使用MySQL 5.0的用户图形界面的输入SQL语句的脚本区，除了可以执行插入操作的SQL语句外，也可以执行修改和删除的SQL语句。

**注意** 在MySQL 5.0的用户图形界面的输入SQL语句的脚本区中，每一个SQL语句需要使用分号（；）进行分隔。

## 12.2 向视图中插入数据记录

与数据表一样，也可以向视图使用INSERT INTO语句插入一条数据记录。对视图中数据的插入操作，最终转化为数据表中数据的插入操作。但是向视图插入一条数据记录时，除了像数据表一样要满足数据约束条件之外，还需要满足一些其他的条件。为了说明这个问题，首先来看一个向视图中插入数据记录的例子。

**例12.5** 向教师视图V\_teacher中插入一条教师信息。

```
INSERT INTO V_teacher
VALUES
('t101234','寒燕',45,'女','计算机系','教授')
```

这段SQL语句表示向教师视图V\_teacher中插入一条教师信息，MySQL 5.0在执行插入的过程中，会得到这样一条错误信息：

```
Field of view 'test_stinfo.v_teacher' underlying table doesn't have a default value
```

这条错误信息提示在执行插入操作时会出现错误。对视图中数据的插入操作，最终转化为数据表中数据的插入操作。使用INSERT INTO语句向教师视图V\_teacher做数据插入操作时，会转化为对教师信息表T\_teacher的插入操作。教师信息表T\_teacher中表示教师所在院校编号的列deptID和表示教师工资的列salary都是被定义为非空约束的列。而这条INSERT INTO语句在向数据表做插入操作时，并没有

插入与这两个列有关的信息，因此在执行插入操作时会出现错误。

从上面的例子中可以看出，并不是所有的视图都可以使用INSERT INTO语句插入数据。在向视图中插入一条数据记录时，还需要满足一些其他的条件。向视图中插入一条数据记录需要满足的基本原则如下：

- ❑ 定义的视图中不能含有DISTINCT关键字、GROUP BY子句以及集合函数。
- ❑ 视图中不能含有使用了表达式的列。例如，视图中有有一个列sumsalary，该列中的数据是由数据表中列salary和列pension中的数据相加得到的。
- ❑ 视图定义中的列需要包含主键列和定义为非空约束的列。

**例12.6** 向学生视图V\_student中插入一条学生记录。

```
INSERT INTO V_student
VALUES
('s286666','李山山',20,'女','19890505');
```

这段SQL语句是向学生视图V\_student中插入一条学生记录。学生视图V\_student中包含了学生信息表中的全部的列，而且学生视图V\_student中没有DISTINCT关键字、GROUP BY子句以及集合函数，也不含有使用了表达式的列。因此该条插入语句可以正确地执行。

为了验证使用INSERT INTO语句是否将学生编号为s286666的学生插入到学生信息表中，可以使用SELECT语句查询学生视图V\_student中的学生信息。其SQL语句如下：

```
SELECT stuID,stuName,age,sex,birth
FROM V_student
```

stuID	stuName	age	sex	birth
s102203	赵亮	23	男	1986-05-16 00:00:00
s112303	郑茹	21	女	1988-01-25 00:00:00
s115263	王海	23	男	1986-08-02 00:00:00
s206363	张明	22	男	1987-04-23 00:00:00
s221256	王昌麟	24	男	1985-03-18 00:00:00
s231456	王玉梅	22	女	1987-03-28 00:00:00
s23256	李玉峰	24	女	1985-08-16 00:00:00
s2532653	李凤	24	女	1985-06-06 00:00:00
s281234	王龙	20	男	1989-02-18 00:00:00
s284321	李霞	20	女	1989-08-20 00:00:00
s286666	李山山	20	女	1989-05-05 00:00:00

这段SQL语句是查询学生视图V\_student中的学生信息。其查询结果如图12.6所示。

从图12.6可以看到，在学生视图V\_student中确实增加了一条学生编号为s286666，学生姓名为李山山的学生记录。

图12.6 查询学生视图V\_student

**注意** 对视图的插入操作，最终转化为数据表的插入操作。

12.3 小结

本章主要介绍了使用INSERT INTO语句向数据表和视图中插入数据的方法。使用INSERT INTO语句不仅可以插入单行数据，也可以使用子查询插入多行数据。在INSERT INTO语句中使用子查询主要是为了实现对数据表中数据的复制。在执行数据表的插入操作时，要注意插入列的顺序、数据类型以及个数要与指定数据表中提供的数据列相一致。

视图中也可以使用INSERT INTO语句插入一条记录，但是在向视图插入数据记录时，还需要有一些额外的限制，需要掌握对视图执行插入操作的基本原则。对视图的插入操作，最终会转化为数据表的插入操作。



## 第13章 修改数据记录

数据操纵语言包括INSERT、UPDATE和DELETE。UPDATE语句主要是用来执行数据的修改操作。使用UPDATE语句既可以修改单行数据，也可以使用子查询修改多行数据；既可以在数据表中修改数据记录，也可以在视图中修改数据记录。但是在视图中修改数据记录时，还需要有一些额外的限制。这一章主要介绍如何使用UPDATE SET语句在数据表和视图中修改数据记录。

本章重点：

- ☐ 修改单行数据记录
- ☐ 在定义有外键约束的表中修改数据记录
- ☐ 修改多行记录
- ☐ 使用子查询修改数据记录
- ☐ 使用CASE条件表达式修改多行记录
- ☐ 利用MySQL 5.0数据库一次修改多条数据记录
- ☐ 在视图中修改数据记录

### 13.1 在数据表中修改数据记录

如果想在数据表中修改数据记录，可以使用UPDATE SET语句。使用UPDATE SET语句既可以在数据表中修改单行数据记录，也可以向数据表中修改多行数据记录，而且利用MySQL 5.0数据库的用户图形界面提供的功能还可以向指定的数据表中一次修改多条数据记录。这一节就来介绍使用UPDATE SET语句在数据表中修改数据的方法。

#### 13.1.1 修改单行数据记录

使用UPDATE SET语句可以在数据表中修改单行数据记录。使用UPDATE SET语句在数据表中修改单行数据记录的语法格式如下：

```
UPDATE table_name
SET column 1= value1[,column 2=v alue2]
WHERE condition
```

其中，UPDATE SET表示在数据表中修改数据记录的关键字；table\_name表示表的名字；关键字SET后面跟的是指定的修改条件，指定的修改条件可以有一个，也可以有多个，多个修改条件之间需要用逗号将其分开。column 1表示指定要修改的列，value1表示要修改的列对应的值；WHERE子句用来指定查询条件。

**例13.1** 修改学生信息表中一条学生记录。

```
UPDATE T_student
```

```
SET age =age+1
WHERE stuID='s281234'
```

这段SQL语句是在学生信息表中将学生编号为s281234的年龄增加1岁。其中，T\_student表示学生信息表；关键字SET后面跟的是指定的修改条件。这里是将表示年龄的列age的值加1；WHERE子句用来指定查询条件，这里指定的查询条件是stuID='s281234'，表示要修改的是学生编号为s281234的年龄。

**注意** 如果数据表中需要修改的值是数字，则可以直接使用数字值；如果数据表中需要修改的值是字符或者是日期类型的数据，则需要使用单引号将其引住。

修改数据的SQL语句正确执行后，可以使用SELECT语句查询教师信息表中的学生信息。其SQL语句如下：

```
SELECT stuID,stuName,age,sex,birth
FROM T_ student
```

这段SQL语句是查询学生信息表中的学生信息。其查询结果如图13.1所示。

从图13.1可以看到，在学生信息表中确实修改了一条学生编号为s281234，学生姓名为王龙的学生记录。该学生的年龄由原来的20岁修改为21岁（可以对比图12.1）。

当然，也可以在UPDATE SET语句中在表名后面指定多个修改条件。例如，在课程信息表中现在要将操作系统的课时增加为80，学分数增加为4。

例13.2 修改课程信息表中操作系统的课时和学分数。

```
UPDATE T_curriculum
SET learnTime =80,credit=4
WHERE curName ='操作系统'
```

这段SQL语句是修改课程信息表中操作系统的课时和学分数，其中T\_curriculum表示课程信息表；关键字SET后面跟的是指定的修改条件，这里指定了两个修改条件，一个是将表示课时的列learnTime的值修改为80，一个是将表示学分的列credit修改为4；WHERE子句用来指定查询条件，curName ='操作系统'用来指定修改课时和学分数的课程是操作系统这门课。

**注意** 使用UPDATE SET语句执行修改操作时，指定的修改条件中要修改的数据必须符合约束条件，要修改的数据必须与指定列的数据类型相匹配。如果修改的数据不符合约束条件，则数据库管理系统会报错，拒绝执行修改操作。

stuID	stuName	age	sex	birth
s102203	赵亮	23	男	1986-05-16 00:00:00
s112303	郑丽	21	女	1988-01-25 00:00:00
s115263	王海	23	男	1986-08-02 00:00:00
s206363	张明	22	男	1987-04-23 00:00:00
s221256	王磊	24	男	1985-03-18 00:00:00
s231456	王玉梅	22	女	1987-03-28 00:00:00
s23256	李玉峰	24	女	1985-09-16 00:00:00
s253263	李凤	24	女	1985-06-06 00:00:00
s281234	王龙	21	男	1989-02-18 00:00:00
s284321	李茜	20	女	1989-08-20 00:00:00
s286666	李山山	20	女	1989-05-05 00:00:00

图13.1 查询学生信息表

13.1.2 在定义有外键约束的表中修改数据记录

在定义有外键约束的表中修改数据记录时，其修改的数据记录需要满足外键约束条件。例如，对于成绩信息表T\_result，该表中定义了一个指向学生信息表的外键约束，其删除方式（ON DELETE）和修改方式（ON UPDATE）都是RESTRICT。如果在成绩信息表T\_result中修改学生编号，该学生编号并不在学生信息表T\_student中，则数据库管理系统会报错，拒绝执行修改操作。

例13.3 在成绩信息表中修改学生编号，该学生编号并不在学生信息表中。

## 零基础学SQL

```
UPDATE T_result
SET stuID = 's111111'
WHERE stuID = 's281234'
```

这段SQL语句是在成绩信息表T\_result中修改学生编号为s281234的学生记录，将该学生的学生编号修改为s111111。由于在学生信息表中不存在学生编号为s111111的学生信息，因此在成绩信息表中执行修改操作时，MySQL数据库管理系统会显示如下的错误信息。

```
Cannot add or update a child row: a foreign key constraint fails ('test_stinfo/t_result',
CONSTRAINT 't_result_ibfk_1' FOREIGN KEY ('stuID') REFERENCES 't_student' ('stuID'))
```

这个错误信息表示数据表之间定义了外键约束关系，修改成绩信息表T\_result中学生编号为s281234的学生记录违反了外键约束条件。在成绩信息表T\_result中已经定义了一个指向学生信息表T\_student的外键约束，由于学生信息表中没有学生编号为s111111的学生记录，因此不能在成绩信息表中将学生编号修改为s111111。

如果希望将成绩信息表中学生编号为s281234的成绩记录信息修改为学生编号为s111111的成绩记录信息，那么需要修改成绩信息表对应的主表学生信息表的学生编号信息。修改学生信息表的SQL语句如下：

```
UPDATE T_student
SET stuID = 's111111'
WHERE stuID = 's281234'
```

这里将学生信息表中学生编号s281234修改为s111111。其修改语句与例13.3的UPDATE语句相同，只需要将表名T\_result换成T\_student即可。如果要想保证修改成功，在成绩信息表T\_result中定义的外键约束中ON UPDATE指定的修改方式必须修改为CASCADE。这样，如果主表学生信息表中的学生编号信息被修改了，那么在从表成绩信息表中，其对应的学生编号也会做相应的修改。

确定成绩信息表T\_result定义的外键约束条件中ON UPDATE指定的修改方式为CASCADE后，其修改学生信息表T\_student的UPDATE语句就可以正常执行。修改学生信息表T\_student的UPDATE语句正常执行后，再查询成绩信息表T\_result，会发现在表T\_result中，学生编号为s281234的成绩记录已经不存在了，原来T\_result表中学生编号为s281234的成绩记录现在都变为了学生编号为s111111的成绩记录了。

在修改学生信息表T\_student的学生记录时，如果在MySQL数据库管理系统中出现了如下的错误信息：

```
Cannot delete or update a parent row: a foreign key constraint fails ('test_stinfo/t2_result',
CONSTRAINT FK_t2_result_1 FOREIGN KEY ('stuID') REFERENCES 't2_student' ('stuID') ON DELETE CASCADE)
```

这个错误信息表示对数据表的修改或者更新操作违反了外键约束条件，此时需要检查成绩信息表T\_result定义的外键约束条件中，其ON UPDATE指定的修改方式是否是CASCADE。如果ON UPDATE指定的修改方式为RESTRICT，则会出现上述错误信息。应该将ON UPDATE指定的修改方式指定为CASCADE。

**例13.4** 将成绩信息表中学生编号为s102203学生选修的t333这门课的成绩加10分。

```
UPDATE T_result
SET result=result+10
WHERE stuID='s102203'
AND curID = 't333'
```

这段SQL语句是在成绩信息表T\_result中修改一条学生编号为s102203的学生成绩记录，将该名学生的课程编号为t333这门课的成绩提高10分。学生编号为s102203的学生记录是在学生信息表中已经存在的学生记录。因此例13.4中的这条修改语句是可以执行的。

修改数据的SQL语句正确执行后，可以使用SELECT语句查询成绩信息表中学生编号为s102203的学生成绩信息。其SQL语句如下：

```
SELECT stuID, curID, result
FROM T_result
WHERE stuID = 's102203'
```

这段SQL语句是查询成绩信息表中的学生编号为s102203，课程编号为t333这门课的成绩。其查询结果如图13.2所示。

stuID	curID	result
s102203	t105	85
s102203	t232	75
s102203	t321	90
s102203	t333	70

从图13.2可以看到，在成绩信息表中学生编号为s102203，课程编号为t333的成绩记录确实已经修改了。（可以对比图8.1）

图13.2 查询成绩信息表学生编号为s102203的学生选修的t333课的成绩

**注意** 在定义有外键约束的表中修改数据记录时，其修改的数据记录需要满足外键约束条件。从表中其ON UPDATE指定的修改方式应该指定为CASCADE。

13.1.3 修改多行记录

使用UPDATE SET语句，也可以在数据表中修改多行数据记录。在使用UPDATE SET语句执行修改操作时，只要在UPDATE SET语句中满足WHERE子句中指定条件的记录，其值都会被修改。

例13.5 修改成绩信息表中学生编号为s281234的成绩信息。

```
UPDATE T_result
SET result=result+10
WHERE stuID='s281234'
```

这段SQL语句是将学生编号为s281234的学生所有的课程成绩都加10分。在成绩信息表T\_result中只要是学生编号为s281234对应的课程，不管有几门课程都会在其课程成绩所对应的列result中将其分数加10，可以使用13.1.2小节中提供的SQL语句查询学生编号为s281234的课程成绩。其查询结果如图13.3所示。

stuID	curID	result
s281234	t105	93
s281234	t232	71
s281234	t321	95
s281234	t333	90

图13.3 查询成绩信息表学生编号为s281234的学生成绩信息

从图13.3可以看到，在成绩信息表中学生编号为s281234对应的课程成绩记录都已经得到修改了。

13.1.4 使用子查询修改数据记录

在UPDATE SET语句中，也可以使用子查询修改数据。使用UPDATE SET语句在数据表中修改数据记录的语法格式如下：

```
UPDATE table_name
SET subquery
WHERE condition
```

## 零基础学SQL

其中，UPDATE SET表示在数据表中修改数据记录的关键字；table\_name表示表的名字；SET后面跟的是指定的修改条件，这里指定的修改条件是一个子查询语句；WHERE子句用来指定查询条件。

**注意** 在UPDATE SET语句中使用子查询在数据表中修改数据时，指定的修改条件中要修改的数据必须符合约束条件，要修改的数据必须与指定列的数据类型相匹配。如果修改的数据不符合约束条件，则数据库管理系统会报错，拒绝执行修改操作。

**例13.6** 修改课程信息表中操作系统这门课的课时和学分，让它与C语言的课时和学分相同。

```
UPDATE T_curriculum
SET (learnTime,credit)=
(SELECT learnTime,credit
FROM T_curriculum
WHERE curName='C语言')
WHERE curName = '操作系统'
```

这段SQL语句是修改课程信息表中操作系统这门课的课时和学分，其指定的修改条件是根据子查询语句中的查询结果来确定的，这里的子查询语句是查询课程名为C语言这门课的课时和学分数。最终要修改的操作系统这门课的课时和学分数是根据子查询中C语言这门课的课时和学分数得来的。

### 13.1.5 使用CASE条件表达式修改多行记录

在10.7.3节中讲到过使用CASE条件表达式实现多重条件分支查询的功能。CASE条件表达式除了可以用于分支查询，也可以将CASE条件表达式应用在UPDATE SET语句中来修改多行记录。下面来看一个使用CASE条件表达式修改多行记录的例子。

这个例子是根据现有教师的工资水平对教师工资进行调整。为了讲解这个例子，这里首先需要做一些准备工作。

(1) 保持原有的教师信息表（T\_teacher）不变，在CREATE TABLE语句中使用子查询将原有的教师信息表（T\_teacher）中教师工资信息复制到新建的T\_teacher\_salary表中。其SQL语句如下：

```
CREATE TABLE T_teacher_salary
AS
(SELECT teaID,teaName,salary
FROM T_teacher
)
```

这段SQL语句使用CREATE TABLE创建一个新的数据表T\_teacher\_salary，在这个数据表中只包含原有的教师信息表（T\_teacher）中的3个列的数据，分别是表示教师编号的列teaID、表示教师姓名的列teaName和表示教师工资的列salary。

(2) 创建完成T\_teacher\_salary数据表之后，再使用SELECT语句查询出T\_teacher\_salary表的教师工资信息。其查询的SQL语句如下：

```
SELECT teaID,teaName,salary
FROM T_teacher_salary
ORDER BY salary
```

这段SQL语句是查询新建的T\_teacher\_salary表中现有教师工资信息，并按照教师工资由低到高的顺序升序排序，其查询结果如图13.4所示。



做完上述准备工作之后，下面就来看如何根据T\_teacher\_salary表中现有的教师工资信息使用CASE条件表达式对教师工资进行修改。

#### 例13.7 修改教师工资（使用CASE条件表达式）。

```
UPDATE T_teacher_salary
SET salary=
CASE
WHEN salary<=3000 THEN salary+salary*0.1
WHEN salary>3000 AND salary<=4000 THEN salary+salary*0.08
WHEN salary>4000 THEN salary+salary*0.05
ELSE salary
END
```

teaID	teaName	salary
t156354	王新	2500
t105320	于波	2800
t102225	赵伟	3000
t186585	孙立	3200
t181585	李慧	3500
t103265	张晶	3800
t106358	毛翠	4000
t156355	李中	4200

图13.4 查询修改之前T\_teacher\_salary表的教师工资信息

这段SQL语句是根据T\_teacher\_salary表中现有教师的工资水平使用CASE条件表达式对教师的工资进行修改。与前面讲过的UPDATE SET修改语句不同，这里的UPDATE SET语句中，在SET salary等号的后面使用的是一个CASE条件表达式，在CASE条件表达式中对现有的教师工资进行判断，如果现有的教师工资低于3000元，就将该教师的工资在原有工资的基础上增加10%；如果现有的教师工资在3000元到4000元之间，就将该教师的工资在原有工资的基础上增加8%；如果现有的教师工资高于4000元，就将该教师的工资在原有工资的基础上增加5%。

使用UPDATE SET语句修改完成T\_teacher\_salary数据表的教师工资之后，再使用SELECT语句查询出T\_teacher\_salary表的教师工资信息。其查询的SQL语句如下：

```
SELECT teaID,teaName,salary
FROM T_teacher_salary
ORDER BY salary
```

这段SQL语句是查询新创建的T\_teacher\_salary表修改后的教师工资信息，并按照教师工资由低到高的顺序升序排序，其查询结果如图13.5所示。

与图13.4中显示的教师工资信息相比，图13.5中的教师工资有了不同程度的变化。例如，王新教师原来的工资是2500元，经过工资调整之后，其工资变成了2750元，比原来的工资增加了10%；孙立教师原来的工资是3200元，经过工资调整之后，其工资变成了3456元，比原来的工资增加了8%；李中教师原来的工资是4200元，经过工资调整之后，其工资变成了4410元，比原来的工资增加了5%。

teaID	teaName	salary
t156354	王新	2750
t105320	于波	3080
t102225	赵伟	3300
t186585	孙立	3456
t181585	李慧	3780
t103265	张晶	4104
t106358	毛翠	4320
t156355	李中	4410

图13.5 查询修改之后T\_teacher\_salary表的教师工资信息

对于上面的例子，如果不使用CASE条件表达式，则需要写3个UPDATE SET语句执行教师工资的修改，而使用CASE条件表达式只需要使用一个UPDATE SET语句就可以完成了。从这个例子中可以看到，使用CASE条件表达式可以根据条件判断中出现的不同情况执行不同的操作，大大提高了程序的执行效率。

### 13.1.6 利用MySQL 5.0数据库一次修改多条数据记录

在MySQL 5.0的用户图形界面中，提供了可以一次执行多条SQL语句的功能。这里介绍一下使用MySQL 5.0用户图形界面一次执行多条修改的SQL语句的使用方法。

(1) 选择“开始”|“所有程序”|“MySQL”菜单，选中“MySQL Query Brower”选项，输入用

用户名、密码，在Default Schema选项对应的文本框中输入一个数据库的名字test\_STInfo，进入到MySQL 5.0的用户图形界面中。

(2) 选择菜单栏中的“File”菜单，在对应的下拉框中选择“New Script Tab”选项。

(3) 选中“New Script Tab”选项后，在MySQL 5.0的用户图形界面会打开一个输入SQL语句的脚本区。在该脚本区中可以输入多条SQL语句，如图13.6所示。

在图13.6中，一共输入了3条SQL语句，这3条SQL语句是在成绩信息表中为学生编号为s281234的学生修改课程成绩信息。这里的每一个SQL语句之间需要用分号（；）将其分开。

在MySQL 5.0的用户图形界面的工具栏中，有一个“Execute”按钮。单击“Execute”按钮可以将脚本区中输入的SQL语句一次全部执行。

使用MySQL 5.0的用户图形界面的输入SQL语句的脚本区，除了可以执行修改操作的SQL语句外，也可以执行插入和删除的SQL语句。

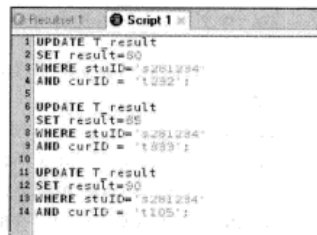


图13.6 SQL语句的脚本区

**注意** 在MySQL 5.0的用户图形界面的输入SQL语句的脚本区中，每一个SQL语句需要使用分号（；）进行分隔。

## 13.2 在视图中修改数据记录

与数据表一样，也可以在视图中使用UPDATE SET语句修改一条数据记录。对视图中数据的修改操作，最终转化为数据表中数据的修改操作。但是在视图中修改一条数据记录时，除了像数据表一样要满足数据约束条件之外，还需要满足一些其他的条件。为了说明这个问题，首先来看一个在视图中修改数据记录的例子。

**例13.8** 修改教师工资视图V\_teacher\_salary中一条教师信息。

```
UPDATE V_teacher_salary
SET maxsalary = maxsalary+500
WHERE dept = '计算机系'
```

这段SQL语句表示修改教师工资视图V\_teacher\_salary中院校为计算机系的教师信息，MySQL 5.0在执行修改的过程中，会得到这样一条错误信息：

```
The target table V_teacher_salary of the UPDATE is not updatable
```

这条错误信息提示在执行修改操作时会出现错误。对视图中数据的修改操作，最终转化为数据表中数据的修改操作。使用UPDATE SET语句向教师工资视图V\_teacher\_salary做数据修改操作时，会转化为对教师信息表T\_teacher的修改操作。由于视图V\_teacher\_salary中的工资是经过聚合函数计算出来的，是对教师信息表T\_teacher中多行记录统计的结果。如果视图中的数据是通过聚合函数计算得来的，那么在使用UPDATE SET语句对视图中的数据进行修改操作时会出现错误。

从上面的例子中可以看出，并不是所有的视图都可以使用UPDATE SET语句修改数据。在视图中如果要修改一条数据记录，还需要满足一些其他的条件。在视图中修改一条数据记录需要满足的基本原则如下：

- ❑ 定义的视图中不能含有DISTINCT关键字、GROUP BY子句以及集合函数。
- ❑ 视图中不能含有使用了表达式的列。例如，视图中有一个列sumsalary，该列中的数据是由数据表中列salary和列pension中的数据相加得到的。

例13.9 在教师视图V\_teacher中修改教师编号为t105320教师的职称。

```
UPDATE V_teacher
SET profession= '副教授'
WHERE teaID = 't105320'
```

这段SQL语句是在教师视图V\_teacher中将教师编号为t105320教师的职称修改为副教授。教师视图V\_teacher包含的列中没有DISTINCT关键字、GROUP BY子句以及集合函数，也不含有使用了表达式的列。因此该条修改语句可以正确地执行。

为了验证使用UPDATE SET语句是否将教师编号为t105320的教师修改到教师信息表中，可以使用SELECT语句查询教师视图V\_teacher中的教师信息。其SQL语句如下：

```
SELECT teaID,teaName,age,sex,,dept,profession
FROM V_teacher
```

这段SQL语句是查询教师视图V\_teacher中的教师信息。其查询结果如图13.7所示。

teaID	teaName	age	sex	dept	profession
t102225	赵伟	38	男	计算机系	副教授
t103265	张磊	43	男	计算机系	教授
t105320	于波	28	男	计算机系	副教授
t106358	毛翠	51	女	计算机系	教授
t156354	王新	33	女	数学系	讲师
t156355	李中	55	女	数学系	教授
t181595	李慧	40	女	物理系	教授
t186585	孙立	48	男	物理系	讲师

图13.7 查询教师视图V\_teacher

从图13.7可以看到，在教师视图V\_teacher中教师编号为t105320教师的职称已经由原来的讲师修改为副教授了。

**注意** 对视图的修改操作，最终转化为数据表的修改操作。

13.3 小结

这一节主要介绍了使用UPDATE SET语句在数据表和视图中修改数据的方法。使用UPDATE SET语句不仅可以修改单行数据，也可以使用子查询修改多行数据。在执行数据表的修改操作时，指定修改条件中要修改的数据必须符合约束条件，要修改的数据必须与指定列的数据类型相匹配。

视图中也可以使用UPDATE SET语句修改记录，但是在视图中修改数据记录时，还需要有一些额外的限制，需要掌握对视图执行修改操作的基本原则。对视图的修改操作，最终会转化为数据表的修改操作。

## 第14章 删除数据记录

数据操纵语言包括INSERT、UPDATE和DELETE。DELETE语句主要是用来执行数据的删除操作。使用DELETE语句既可以删除满足条件的数据，也可以使用子查询删除指定条件的数据；既可以在数据表中删除数据记录，也可以在视图中删除数据记录。但是在视图中删除数据记录时，还需要有一些额外的限制。这一章就主要介绍如何使用DELETE语句在数据表和视图中删除数据记录。

本章重点：

- ☐ 删除满足条件的数据记录
- ☐ 在定义有外键约束的表中删除数据记录
- ☐ 使用子查询删除指定条件的数据记录
- ☐ 利用MySQL 5.0数据库一次删除多条数据记录
- ☐ 删除数据表中所有记录
- ☐ 使用TRUNCATE关键字删除数据表中记录
- ☐ 在视图中删除数据记录

### 14.1 使用DELETE语句删除数据记录

如果想在数据表中删除数据记录，可以使用DELETE语句。使用DELETE语句既可以在数据表中删除满足条件数据记录，也可以使用子查询删除指定条件的数据，而且利用MySQL 5.0数据库的用户图形界面提供的功能还可以在指定的数据表中一次删除多条数据记录。如果数据表中的记录都不需要了，也可以使用DELETE语句将数据表中所有的记录都删除掉。这一节就来介绍使用DELETE语句在数据表中删除数据的方法。

#### 14.1.1 删除满足条件的数据记录

使用DELETE语句可以在数据表中删除满足条件的数据记录。使用DELETE语句在数据表中删除满足条件数据记录的语法格式如下：

```
DELETE FROM table_name
WHERE condition
```

其中，DELETE FROM表示在数据表中删除数据记录的关键字；table\_name表示表的名字；WHERE子句后的condition用来指定查询限制条件。

**注意**

使用DELETE语句执行数据表的删除操作时，不需要指定数据列的名字，因为DELETE语句执行的操作是删除数据表中某一行的记录，而不是删除某一个单独的列的记录。

**例14.1** 删除学生信息表中学生编号为s286666的学生记录。

```
DELETE FROM T_student
WHERE stuID='s286666'
```

这段SQL语句是在学生信息表中将学生编号为s286666的学生记录删除掉。其中，T\_student表示学生信息表；WHERE子句用来指定查询条件，这里指定的查询条件是stuID='s286666'，表示要删除的是学生编号s286666的学生记录。

**注意** 如果数据表中需要删除的值是数字，则可以直接使用数字值；如果数据表中需要删除的值是字符或者是日期类型的数据，则需要使用单引号将其引住。

删除数据的SQL语句正确执行后，可以使用SELECT语句查询教师信息表中的学生信息。其SQL语句如下：

```
SELECT stuID,stuName,age,sex,birth
FROM T_student
```

这段SQL语句是查询学生信息表中的学生信息。其查询结果如图14.1所示。

从图14.1可以看到，在学生信息表中确实删除了一条学生编号为s286666的学生记录（可以对比图13.1）。

stuID	stuName	age	sex	birth
s102203	赵亮	23	男	1986-05-16 00:00:00
s112303	郑茹	21	女	1988-01-25 00:00:00
s115263	王海	23	男	1986-08-02 00:00:00
s206363	张明	22	男	1967-04-23 00:00:00
s221256	王昌辉	24	男	1985-03-18 00:00:00
s231456	王玉梅	22	女	1987-03-28 00:00:00
s23256	李玉峰	24	女	1985-08-16 00:00:00
s2532653	李凤	24	女	1985-06-06 00:00:00
s261234	王龙	21	男	1989-02-18 00:00:00
s284321	李霞	20	女	1989-08-20 00:00:00

图14.1 查询学生信息表

14.1.2 在定义有外键约束的表中删除数据记录

在定义有外键约束的表中删除数据记录时，其删除的数据记录需要满足外键约束条件。例如，对于成绩信息表T\_result，该表中定义了一个指向学生信息表的外键约束，其删除方式（ON DELETE）和修改方式（ON UPDATE）都是RESTRICT。如果在学生信息表T\_result中删除学生编号，则数据库管理系统会报错，拒绝执行删除操作。

例14.2 在学生信息表中删除学生编号。

```
DELETE FROM T_student
WHERE stuID='s206363'
```

这段SQL语句是在学生信息表T\_student中删除学生编号为s206363的学生记录。在学生信息表中执行删除操作时，MySQL数据库管理系统会显示如下的错误信息。

```
Cannot delete or update a parent row: a foreign key constraint fails ('test_stinfo/t2_result',
CONSTRAINT 'FK_t2_result_1' FOREIGN KEY ('stuID') REFERENCES 't2_student' ('stuID'))
```

这个错误信息表示数据表之间定义了外键约束关系，删除主表学生信息表T\_student中学生编号为s206363的学生记录违反了外键约束条件。

如果希望将主表学生信息表中学生编号为s206363的记录信息删除，在成绩信息表T\_result中定义的外键约束中ON DELETE指定的删除方式必须为CASCADE。这样，如果主表学生信息表中的学生编号信息被删除了，那么在从表成绩信息表中，其对应的学生编号也会做相应的删除。

确定成绩信息表T\_result定义的外键约束条件中ON DELETE指定的删除方式为CASCADE后，其删除学生信息表T\_student的DELETE语句就可以正常执行。删除学生信息表T\_student的DELETE语句正常执行后，再查询成绩信息表T\_result，会发现在表T\_result中，学生编号为s206363的成绩记录已经不存在了。其从表成绩信息表查询的结果如图14.2所示。



## 零基础学SQL

在删除学生信息表T\_student的学生记录时，如果在MySQL数据库管理系统中出现了如下的错误信息：

```
Cannot delete or update a parent row: a foreign key constraint fails ('test_stinfo/t2_result',  
CONSTRAINT FK_t2_result_1' FOREIGN KEY ('stuID') REFERENCES 't2_student' ('stuID') ON DELETE CASCADE)
```

这个错误信息表示对数据表的删除或者更新操作违反了外键约束条件，此时需要检查成绩信息表T\_result定义的外键约束条件中，其ON DELETE指定的删除方式是否是CASCADE。如果ON DELETE指定的删除方式为RESTRICT，则会出现上述错误信息。应该将ON DELETE指定的删除方式指定为CASCADE。

如果多个表之间都定义了外键约束条件，使用DELETE语句执行删除操作时，也可以按照从最内层的子表开始，一直到最外层主表的顺序删除数据。例如，对于成绩信息表T\_result和学生信息表T\_student来说，如果希望删除学生编号为s206363的成绩记录信息和学生信息，可以先在成绩信息表中将该学生的成绩记录删除，然后再到学生信息表中将学生编号为s206363的学生信息删除。

stuID	curID	result
s102203	i105	85
s102203	i232	75
s102203	i321	90
s102203	i333	60
s111111	i105	93
s111111	i232	71
s111111	i321	95
s111111	i333	90
s221256	i232	80
s2532653	i105	90
s2532653	i232	70
s2532653	i321	90
s2532653	i333	53

图14.2 查询执行完删除操作的成绩信息表

**注意** 在定义有外键约束的表中删除数据记录时，其删除的数据记录需要满足外键约束条件。从表中其ON DELETE指定的删除方式应该指定为CASCADE。

### 14.1.3 使用子查询删除指定条件的数据记录

在DELETE语句中，也可以使用子查询删除数据。使用DELETE语句在数据表中删除指定条件的数据记录的语法格式如下：

```
DELETE table_name  
WHERE subquery
```

其中，DELETE表示在数据表中删除数据记录的关键字；table\_name表示表的名字；WHERE子句用来指定查询条件，这里指定查询条件可以是一个子查询语句。

**例14.3** 删除成绩信息表中学生编号为s102203选修的数据库基础课的成绩。

```
DELETE FROM T_result  
WHERE curID=  
(SELECT C.curID  
FROM T_curriculum C  
WHERE C.curName = '数据库基础')  
AND stuID='s102203'
```

这段SQL语句是删除成绩信息表中学生编号为s102203选修的数据库基础课的成绩。其中，T\_result表示成绩信息表的表名。由于在成绩信息表中只有课程的编号，没有课程的名字，因此想要删除学生编号为s102203选修的数据库基础课的成绩，就需要取得数据库基础课对应的课程编号。这里在WHERE子句中使用一个子查询语句查询课程信息表T\_curriculum中数据库基础课对应的课程编号，并将查询到的课程编号作为DELETE语句中WHERE子句的一个删除条件。WHERE子句后的AND子句用来指定学生的编号。

### 14.1.4 利用MySQL 5.0数据库一次删除多条数据记录

在MySQL 5.0的用户图形界面中，提供了可以一次执行多条SQL语句的功能。这里介绍一下使用MySQL 5.0用户图形界面一次执行多条删除的SQL语句的使用方法。

(1) 单击“开始”|“所有程序”|“MySQL”|“MySQL Query Brower”命令，输入用户名、密码，在Default Schema选项对应的文本框中输入一个数据库的名字test\_STInfo，进入到MySQL 5.0的用户图形界面中。

(2) 选择菜单栏中的“File”菜单，在对应的下拉框中选择“New Script Tab”选项。

(3) 选中“New Script Tab”选项后，在MySQL 5.0的用户图形界面会打开一个输入SQL语句的脚本区。在该脚本区中可以输入多条SQL语句，如图14.3所示。

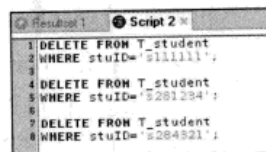


图14.3 SQL语句的脚本区

在图14.3中，一共输入了3条SQL语句，这3条SQL语句是分别是要删除学生编号为s111111、学生编号为s281234和学生编号为s284321的学生信息。这里的每一个SQL语句之间需要用分号（；）将其分开。

在MySQL 5.0的用户图形界面的工具栏中，有一个“Execute”按钮。单击“Execute”按钮可以将脚本区中输入的SQL语句一次全部执行。

使用MySQL 5.0的用户图形界面的输入SQL语句的脚本区，除了可以执行删除操作的SQL语句外，也可以执行插入和修改的SQL语句。

**注意** 在MySQL 5.0的用户图形界面的输入SQL语句的脚本区中，每一个SQL语句需要使用分号（；）进行分隔。

### 14.1.5 删除数据表中所有记录

如果在DELETE语句中，不使用WHERE子句限制删除条件，则表示将指定数据表中的所有数据记录全部删除掉。其语法格式如下：

```
DELETE FROM table_name
```

其中，table\_name表示数据表的名字。使用上面的DELETE语句，会将指定数据表中的所有记录全部删除，但是数据表仍然保留，即把数据表中的数据记录清空。

**例14.4** 删除课程信息表中全部记录。

```
DELETE FROM T_curriculum
```

**注意** 如果在DELETE语句中没有使用WHERE子句限定删除条件，则会删除数据表中的所有记录。所以在使用DELETE语句时，需要慎重考虑是否要将数据表中的记录全部删除。

## 14.2 使用TRUNCATE语句删除数据表中所有记录

在14.1.5小节中介绍了使用不带WHERE子句的DELETE语句删除数据表中所有记录的方法，除了可以使用不带WHERE子句的DELETE语句删除数据表中所有记录，还可以使用TRUNCATE语句来删除

## 零基础学SQL

数据表中所有记录。其语法规则如下：

```
TRUNCATE TABLE table_name
```

其中，TRUNCATE TABLE是表示删除数据表中所有记录的关键字；table\_name表示要删除的数据表的名字。

**例14.5** 删除T\_teacher\_salary表中全部记录。

```
TRUNCATE TABLE T_teacher_salary
```

与不带WHERE子句的DELETE语句的作用相同，该条SQL语句也是将T\_teacher\_salary表中的记录全部删除，但是会保留T\_teacher\_salary表的结构及其定义的约束条件。

虽然功能上与不带WHERE子句的DELETE语句的功能相同，但是它与DELETE语句还是有些区别的。

- DELETE语句在删除一个数据行之前，需要在事务处理日志中记录相关操作，一旦删除失败，可以通过ROLLBACK（回滚），而TRUNCATE语句在删除数据表时不会操作事务处理日志中的记录，一旦删除失败，也不会执行ROLLBACK（回滚）操作。
- DELETE语句是对数据记录一行一行地执行删除操作，删除的速度较慢；而TRUNCATE语句在删除数据表时是一次性将与数据表有关的所有记录删除，其删除速度要比使用DELETE语句删除记录的速度快。

### 14.3 在视图中删除数据记录

与数据表一样，也可以在视图使用DELETE语句删除一条数据记录。对视图中数据的删除操作，最终转化为数据表中数据的删除操作。但是在视图中删除一条数据记录时，除了像数据表一样要满足数据约束条件之外，还需要满足一些其他的条件。为了说明这个问题，首先来看一个在视图中删除数据记录的例子。

**例14.6** 删除教师工资视图V\_teacher\_salary中一条教师信息。

```
DELETE FROM V_teacher_salary  
WHERE dept = '计算机系'
```

这段SQL语句表示删除教师工资视图V\_teacher\_salary中院校为计算机系的教师信息，MySQL 5.0在执行删除的过程中，会得到这样一条错误信息：

```
The target table V_teacher_salary of the DELETE is not updatable
```

这条错误信息提示在执行删除操作时会出现错误。对视图中数据的删除操作，最终转化为数据表中数据的删除操作。使用DELETE语句向教师工资视图V\_teacher\_salary做数据删除操作时，会转化为对教师信息表T\_teacher的删除操作。由于视图V\_teacher\_salary中的工资是经过聚合函数计算出来的，是对教师信息表T\_teacher中多行记录统计的结果。如果视图中的数据是通过聚合函数计算得来的，那么在使用DELETE语句对视图中的数据进行删除操作时会出现错误。

从上面的例子中可以看出，并不是所有的视图都可以使用DELETE语句删除数据。在视图中如果要删除一条数据记录，还需要满足其他的条件。在视图中删除一条数据记录需要满足的基本原则为：定义的视图中不能含有DISTINCT关键字、GROUP BY子句以及集合函数。

**例14.7** 在教师视图V\_teacher中删除教师编号为t105320教师信息。

```
DELETE FROM V_teacher
WHERE teaID = 't105320'
```

这段SQL语句是在教师视图V\_teacher中将教师编号为t105320教师信息删除掉。教师视图V\_teacher包含的列中没有DISTINCT关键字、GROUP BY子句以及集合函数。因此该条删除语句可以正确地执行。

为了验证使用DELETE语句是否将教师编号为t105320教师信息从教师信息表中删除了，可以使用SELECT语句查询教师视图V\_teacher中的教师信息。其SQL语句如下：

```
SELECT teaID,teaName,age,sex,,dept,profession
FROM V_teacher
```

这段SQL语句是查询教师视图V\_teacher中的教师信息。其查询结果如图14.4所示。

teaID	teaName	age	sex	dept	profession
t102225	赵伟	38	男	计算机系	副教授
t103285	张磊	43	男	计算机系	教授
t106358	毛翠	51	女	计算机系	教授
t156354	王新	33	女	数学系	讲师
t156355	李中	55	女	数学系	教授
t181585	李慧	40	女	物理系	教授
t186585	孙立	48	男	物理系	讲师

图14.4 查询教师视图V\_teacher

从图14.4可以看到，在教师视图V\_teacher中教师编号为t105320的教师信息在教师信息表中已经不存在了。（可以对比图13.7）

**注意** 对视图的删除操作，最终转化为数据表的删除操作。

14.4 小结

本章主要介绍了使用DELETE语句在数据表和视图中删除数据的方法。使用DELETE语句不仅可以删除数据表中满足条件的数据记录，也可以使用子查询删除数据表中指定条件的数据记录。如果要想删除数据表中的所有记录，除了使用DELETE语句之外，还可以使用TRUNCATE语句删除数据表中所有记录。

视图中也可以使用DELETE语句删除记录，但是在视图中删除数据记录时，还需要有一些额外的限制，需要掌握对视图执行删除操作的基本原则。对视图的删除操作，最终会转化为数据表的删除操作。

## 第五篇

# 数据控制

### 第15章 权限的授予与回收

在SQL语句中，使用GRANT语句可以为指定用户授予相应的操作权限，使用REVOKE可以回收权限。这一章主要介绍使用GRANT语句授予权限和使用REVOKE语句回收权限的方法。另外，数据库管理系统中都提供了可以为用户授予权限的操作，这一章也以MySQL 5.0数据库为例，介绍在MySQL 5.0数据库中使用Adimistrator管理系统授予用户权限的方法。

本章重点：

- ☐ 数据库及其不同对象允许的操作权限
- ☐ 为指定用户授予操作数据表的权限
- ☐ 为指定用户授予操作数据列的权限
- ☐ 为指定用户授予授权其他用户的权限
- ☐ 为用户授予创建数据库的权限
- ☐ 将操作权限授予所有用户
- ☐ 使用Adimistrator管理系统授予用户权限
- ☐ 权限的回收

#### 15.1 数据库及其不同对象允许的操作权限

数据库对象包括数据表、视图、索引等。在使用SQL语句执行权限授予操作时，不同的对象允许的操作权限也不完全相同。

- ☐ 数据表：SELECT、INSERT、UPDATE、DELETE、ALTER、INDEX、ALLPRIVIEGES。
- ☐ 视图：SELECT、INSERT、UPDATE、DELETE、ALLPRIVIEGES。
- ☐ 数据列：SELECT、INSERT、UPDATE、DELETE、ALLPRIVIEGES。
- ☐ 数据库：CREATETAB。

其中，SELECT表示查询权限；INSERT表示增加数据权限；UPDATE表示修改数据权限；DELETE表示删除数据权限；ALTER表示修改数据表权限；INDEX表示索引权限；ALLPRIVIEGES表示全部操作权限。



如果现在有3个用户和3个数据库，3个用户的用户名分别为user1、user2和user3，对应的3个数据库的名字分别为test\_1、test\_2和test\_3。其中，数据库test\_1中包含有一个教师信息表T\_teacher；数据库test\_2中包含有学生信息表T\_student、课程信息表T\_curriculum和成绩信息表T\_result；数据库test\_3中包含有院校信息表T\_dept。现在想要为这3个用户分配不同的操作权限，应该如何操作。在15.2节中将介绍权限授予的方法。

## 15.2 授予权限

在SQL语句中可以使用GRANT语句为指定用户授予不同的权限。这一节以数据表为例，介绍使用SQL语句授予权限的方法（视图的操作方法与数据表相同）。另外，不同的数据库管理系统也提供了授予用户权限的方法，这里以MySQL 5.0数据库为例，介绍如何使用MySQL 5.0数据库的Adimistrator管理系统授予用户权限。

### 15.2.1 授予指定用户操作数据表的权限

使用GRANT语句可以为指定用户授予操作数据表的权限。使用GRANT语句为指定用户授予操作数据表的权限的语法格式如下：

```
GRANT 权限[,权限] ON TABLE 表名[,表名]
TO 用户[,用户]
```

其中，GRANT关键字表示执行授予权限的操作；GRANT关键字后面跟的是要为数据表授予的权限，包括SELECT、INSERT、UPDATE、DELETE、ALTER、INDEX、ALLPRIVIEGES，多个权限之间需要用逗号分开；关键字ON TABLE表示要为数据表授予权限；关键字ON TABLE后面跟的是表的名字，表的名字可以有多个，多个表名之间需要用逗号分开；关键字TO后面跟的是用户名，表示将权限授予给哪个用户，用户名也可以有多个，多个用户名之间需要用逗号分开。

例15.1 将查询数据库test\_1中教师信息表的权限授予user2。

```
GRANT SELECT ON TABLE test_1.T_teacher
TO user2
```

这段SQL语句是将查询数据库test\_1中教师信息表的权限授予user2。其中，SELECT表示查询权限；test\_1.T\_teacher表示要查询的数据表是数据库test\_1中教师信息表；user2表示授予权限的用户名。

例15.2 将更新数据库test\_1中教师信息表的权限授予user2和user3。

```
GRANT INSERT,UPDATE,DELETE ON TABLE test_1.T_teacher
TO user2, user3
```

这段SQL语句是将更新数据库test\_1中教师信息表的权限授予user2和user3。这里所说的更新包括对数据表中数据的增加、删除和修改操作。user2和user3表示授予权限的用户名。

例15.3 将数据库test\_2中学生信息表、课程信息表和成绩信息表的全部权限授予user2。

```
GRANT ALLPRIVIEGES ON TABLE test_2.T_student, test_2T_curriculum, test_2T_result
TO user2
```

这段SQL语句是将数据库test\_2中学生信息表、课程信息表和成绩信息表的全部权限授予user2。这里的ALLPRIVIEGES表示对数据表操作的全部权限，包括SELECT、INSERT、UPDATE、DELETE、ALTER、INDEX。

## 15.2.2 授予指定用户操作数据列的权限

使用GRANT语句可以为指定用户授予操作数据列的权限。使用GRANT语句为指定用户授予操作数据列的权限的语法格式如下：

```
GRANT 权限[,权限] ON TABLE 表名[,表名]
TO 用户[,用户]
```

其中，GRANT关键字表示执行授予权限的操作；GRANT关键字后面跟的是要为数据列授予的权限，包括SELECT、INSERT、UPDATE、DELETE、ALLPRIVIEGES，多个权限之间需要用逗号分开；关键字ON TABLE表示要为数据表授予权限；关键字ON TABLE后面跟的是表的名字，表的名字可以有多个，多个表名之间需要用逗号分开；关键字TO后面跟的是用户名，表示将权限授予给哪个用户。

例15.4 将修改数据库test\_1中教师信息表教师工资和津贴的权限授予user3。

```
GRANT UPDATE(salary,pension) ON TABLE test_1. T_teacher
TO user3
```

这段SQL语句是将修改数据库test\_1中教师信息表中教师工资和津贴的权限授予user3。其中，UPDATE(salary, pension)表示修改教师工资和津贴的权限；test\_1. T\_teacher表示要修改的数据表是数据库test\_1中教师信息表；user3表示授予权限的用户名。

## 15.2.3 授予指定用户授权的权限

如果获得授权的用户希望将得到的授权再授予其他的用户，则需要在GRANT语句后加上一个WITH GRANT OPTION。其语法格式如下：

```
GRANT 权限[,权限] ON TABLE 表名[,表名]
TO 用户[,用户]
WITH GRANT OPTION
```

其中，GRANT关键字表示执行授予权限的操作；GRANT关键字后面跟的是要为数据列授予的权限，包括SELECT、INSERT、UPDATE、DELETE、ALLPRIVIEGES，多个权限之间需要用逗号分开；关键字ON TABLE表示要为数据表授予权限；关键字ON TABLE后面跟的是表的名字，表的名字可以有多个，多个表名之间需要用逗号分开；关键字TO后面跟的是用户名，表示将权限授予给哪个用户；WITH GRANT OPTION表示获得指定权限的用户有权将该权限再授予其他的用户。

例15.5 将查询数据库test\_1中教师信息表的权限授予user2，并允许user2将该授权授予其他用户。

```
GRANT SELECT ON TABLE test_1. T_teacher
TO user2
WITH GRANT OPTION
```

这段SQL语句中，user2不仅拥有了查询数据库test\_1中教师信息表的权限，还有权将该查询权限授予给其他的用户。例如，user2可以将该权限授予给user3。

```
GRANT SELECT ON TABLE test_1. T_teacher
TO user3
```

这里的GRANT语句中，并没有使用WITH GRANT OPTION，因此user3就没有权利将查询数据库test\_1中教师信息表的权限再授予其他用户。如果希望user3也可以将查询数据库test\_1中教师信息表的权限授予其他用户，可以在上面的GRANT语句的TO user3后面，添加WITH GRANT OPTION。

### 15.2.4 授予创建数据表的权限

使用GRANT语句可以为指定用户授予创建数据表的权限。使用GRANT语句为指定用户创建数据表的权限的语法格式如下：

```
GRANT CREATETAB ON DATABASE DBname  
TO 用户
```

其中，GRANT关键字表示执行授予权限的操作；GRANT关键字后面跟的CREATETAB表示创建数据库的权限；关键字ON DATABASE后面跟的DBName是数据库的名字；关键字TO后面跟的是用户名，表示将在句为DBName数据库中创建数据表的权限授予给哪一个用户。

例15.6 将创建数据库的权限授予user3。

```
GRANT CREATETAB ON DATABASE test_4  
TO user3
```

这段SQL语句是将在数据库test\_4中创建数据表的权限授予user3。其中，test\_4表示数据库的名字；user3表示授予权限的用户名。

### 15.2.5 将操作权限授予所有用户

使用GRANT语句可以将操作权限授予所有用户。使用GRANT语句将操作权限授予所有用户的语法格式如下：

```
GRANT 权限[,权限] ON TABLE 表名[,表名]  
TO PUBLIC
```

其中，GRANT关键字表示执行授予权限的操作；GRANT关键字后面跟的是要为数据列授予的权限，包括SELECT、INSERT、UPDATE、DELETE、ALLPRIVIEGES，多个权限之间需要用逗号分开；关键字ON TABLE表示要为数据表授予权限；关键字ON TABLE后面跟的是表的名字，表的名字可以有多个，多个表名之间需要用逗号分开；关键字TO后面跟的是PUBLIC，表示将权限授予所有用户。

例15.7 将更新数据库test\_1中教师信息表的权限授予所有用户。

```
GRANT INSERT,UPDATE,DELETE ON TABLE test_1. T_teacher  
TO PUBLIC
```

这段SQL语句是将更新数据库test\_1中教师信息表的权限授予所有用户。其中，PUBLIC表示所有合法的用户都会得到更新数据库test\_1中教师信息表的权限。

### 15.2.6 使用Administrator管理系统授予用户权限

除了可以使用GRANT语句为指定用户授予不同的权限外，在数据库系统也提供了可以设置操作权限的方法。这里以MySQL 5.0数据库为例，讲解如何在MySQL 5.0数据库中使用Administrator管理系统授予用户权限。

(1) 选择“开始”|“所有程序”|“MySQL”|“MySQL Administrator”命令，如图15.1所示。

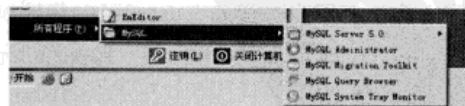


图15.1 MySQL 5.0文件内容

(2) 在出现的“MySQL Administrator”登录界面中输入用户名和密码。这里输入的用户名和密码都为root，如图15.2所示。

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

(3) 进入到“MySQL Administrator”界面后，选择“User Administrator”选项，如图15.3所示。

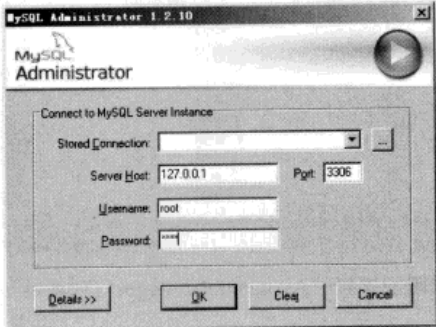


图15.2 在MySQL Administrator登录界面输入用户名和密码

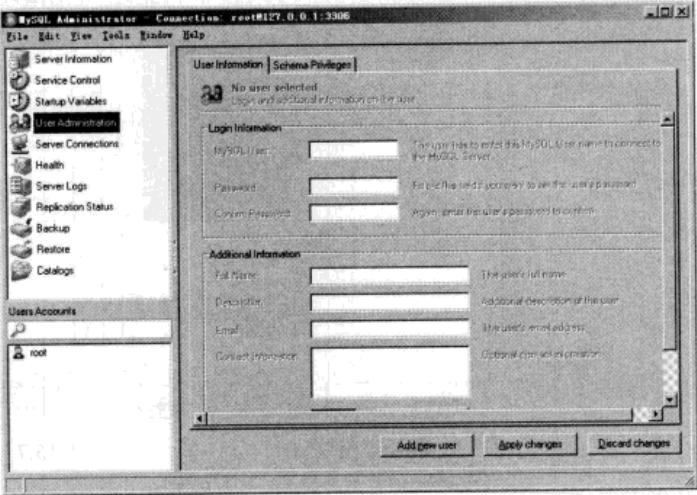


图15.3 MySQL Administrator界面

(4) 单击“Add new user”按钮，在“User Information”选项卡下的文本框中输入用户名和密码，这里输入的用户名和密码为user1，如图15.4所示。

(5) 输入用户名和密码后，选择“Schema Privileges”选项卡，在“Schema”选项下面对应的是MySQL 5.0中所有的数据库信息。这里选中test\_1数据库，在“Available Privileges”选项的下面包含有该数据库的所有权限，如图15.5所示。

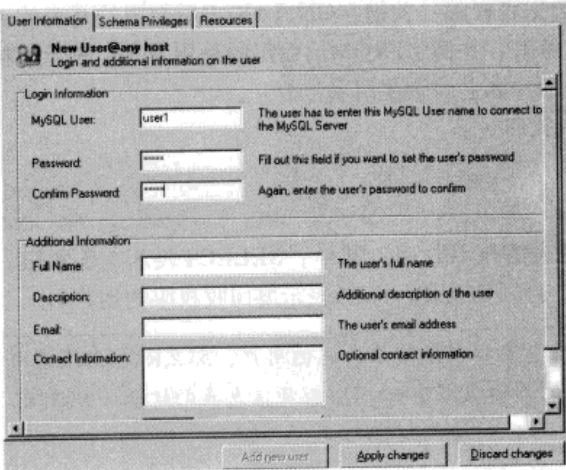


图15.4 输入用户名和密码

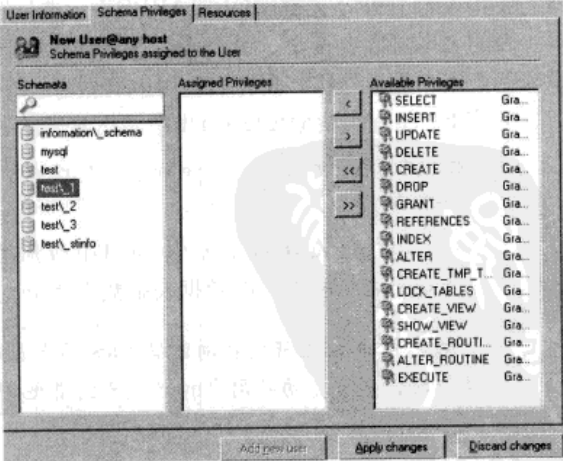


图15.5 选择test\_1数据库

(6) 选中“Available Privileges”列表中需要的权限，如图15.6所示。

(7) 单击<图标，将选中的权限放置到“Assigned Privileges”列表中，如图15.7所示。

(8) 单击界面下方的“Apply changes”按钮，完成权限的授予。

零基础学SQL

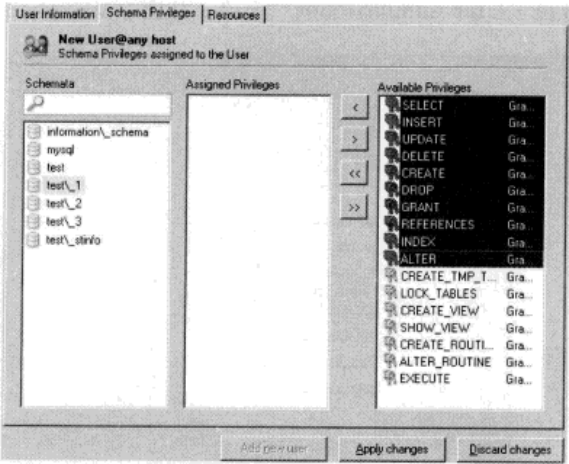


图15.6 选择可用权限

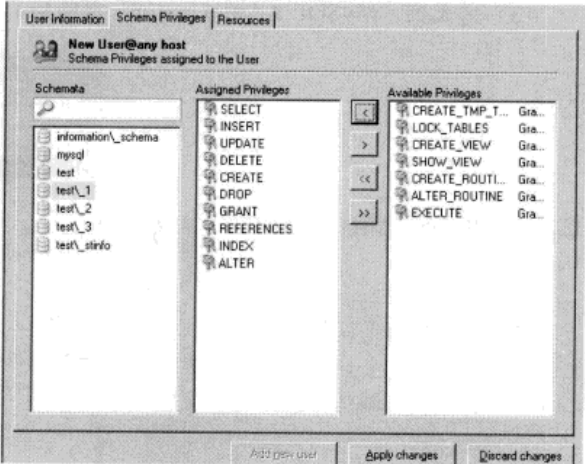


图15.7 将权限放置到“Assigned Privileges”列表中

15.3 回收权限

如果想将授予的权限收回，可以使用REVOKE语句回收权限。使用REVOKE语句回收权限的语法规则如下：

```
REVOKE 权限[, 权限] ON TABLE 表名[, 表名]
FROM 用户[, 用户]
```

其中，REVOKE关键字表示执行回收权限的操作；REVOKE关键字后面跟的是要回收数据表的权限，包括SELECT、INSERT、UPDATE、DELETE、ALTER、INDEX、ALLPRIVIEGES，多个权限之间需要用逗号分开；关键字ON TABLE表示要回收数据表权限；关键字ON TABLE后面跟的是表的名字，表的名字可以有多个，多个表名之间需要用逗号分开；关键字FROM后面跟的是用户名，表示将权限从哪个用户那里回收，用户名也可以有多个，多个用户名之间需要用逗号分开。

例15.8 将查询数据库test\_1中教师信息表的权限收回。

```
REVOKE SELECT ON TABLE test_1. T_teacher
FROM user2
```

这段SQL语句是将查询数据库test\_1中教师信息表的权限收回。其中，SELECT表示查询权限；test\_1. T\_teacher表示要查询的数据表是数据库test\_1中教师信息表；user2表示要回收权限的用户名。

**注意** 如果用户user2还将该查询数据库test\_1中教师信息表的权限授予了其他用户，那么REVOKE语句执行时，也会自动将用户user2授予给其他用户的查询数据库test\_1中教师信息表的权限一并收回。

15.4 小结

本章主要以数据表为例，介绍了使用GRANT语句授予权限和使用REVOKE语句回收权限的方法。另外，不同的数据库系统也提供了可以设置操作权限的方法。也可以通过Administrator管理系统实现权限的授予和回收。



## 第16章 事务的控制与管理

在实际对数据库的使用中，会出现多个用户同时对某一个数据表进行访问或者修改的情况。当多个用户在同一时间对同一张数据表都进行读取或者修改操作时，若处理不当，就可能会导致数据发生冲突的问题。例如，两个人在同一城市的不同地点同时都想买一张飞往大连的飞机票，而在联网的售票系统中，飞往大连的飞机票只有一张，如果售票系统不够完善，就会出现两个人购买了同一航班的同一座位的机票，这样的事情显然是不希望发生的。为了解决类似这样的问题，就需要使用事务的控制和管理机制。SQL语言也支持使用事务。本章就来介绍与事务控制和管理有关的内容。

本章重点：

- ☐ 事务的概念及其ACID属性
- ☐ 事务的控制
- ☐ 并发事务的工作流程
- ☐ 并发事务中存在的问题
- ☐ 事务的隔离级别
- ☐ 设置隔离级别和事务访问模式的方法

### 16.1 事务的概念

事务是指单个逻辑工作单元执行的操作的集合。通过事务处理，保证了数据库中数据的一致性。事务需要满足ACID属性，即Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）和Durability（持久性）。本节就来介绍事务的这4个基本属性。

#### 16.1.1 原子性

所谓事务的原子性，是指同一个事务中所有执行的操作，要么全部成功，要么全都不会执行，即会退回到这个操作执行之前的状态。例如，有两个用户，用户user1持有一个账户A，用户user2持有一个账户B，在数据库中，有一个用来存储账户信息的表accountTable，在这个数据表中保存有两个列，列cashValue表示账户中存储的现金数，列accoutUser表示指定的账户信息。现在用户user1希望将账户A中的1000元钱转账给用户user2持有的账户B中，这个操作需要分为以下两个步骤：

(1) 修改账户A中的现金数。将账户A中的现金数减掉1000，其SQL语句如下：

```
UPDATE accountTable
SET cashValue = cashValue-1000
WHERE accoutUser = 'A'
```

(2) 修改账户B中的现金数。将账户B中的现金数增加1000，其SQL语句如下：

```
UPDATE accountTable
```

```
SET cashValue = cashValue+1000  
WHERE accoutUser = 'B'
```

如果在执行第一个SQL语句之后，第二个SQL语句之前，突然断电了，那么即使第一个操作被执行了，也会被撤销，会回到第一个SQL语句执行操作之前的状态。也就是说，上面的两个操作都不会被执行，账户A中的现金数不会减少，账户B中的现金数也不会增加。

### 16.1.2 一致性

所谓事务的一致性，是指一个事务操作执行完成之后，数据库中数据必须处于合法一致的状态中。如果事务在执行时，数据库中的数据没有保持合法一致的状态，即出现了非法的数据，那么数据库管理系统就会把数据库恢复到该事务执行之前的那个合法的状态中。例如，用户user1只有一个账户A，账户A中共有5000元钱，用户user2持有一个账户B，账户B中也有5000元钱，账户A和账户B中一共有10000元钱，现在用户user1将账户A中的1000元钱转账给用户user2持有的账户B中，当转账操作执行完成之后，账户A中应该还有4000元钱，账户B中应该有6000元钱。账户A和账户B在转账完成之后余款的总和还应该是10000元钱。也就是说，当执行完成这个交易操作之后，数据库从一个合法状态转到了另外一个合法状态中。两个合法状态中账户A和账户B的余款总和保持一致。

### 16.1.3 隔离性

所谓事务的隔离性，是指事务看到的数据库中数据要么是这个事务被修改之前的状态，要么是这个事情被修改之后的状态。多个事务可以并发执行，在执行过程中彼此不会受到影响。

事务的隔离性为事务的并发操作中数据的安全性提供了保证。但是隔离级别与数据库的并发性是成反比的。隔离级别越低，数据库的并发性就越好，访问速度越快；隔离级别越高，会导致数据库的并发性越差，访问速度越慢。SQL92中定义了4种不同的隔离级别，不同的隔离级别在解决并发事务时会出现不同的结果。不同的数据库系统也有不同的默认隔离级别。有关事务的4种隔离级别将在16.3.3节中介绍。

### 16.1.4 持久性

所谓事务的持久性，是指如果一个事务被成功地修改，其结果在数据库中不会因为软件、硬件、系统等故障而改变，也不会因为数据库中的其他操作而受到影响。也就是说，事务一旦成功提交，在数据库中的数据就会永久地保持下来。

## 16.2 控制事务

事务控制包括START TRANSACTION、COMMIT、ROLLBACK等语句。使用START TRANSACTION或者BEGIN等语句可以开始一个事务；使用COMMIT语句可以提交一个事务；使用ROLLBACK语句可以回滚一个事务。这一节就介绍这几种控制事务的方法。

### 16.2.1 开始事务

开始一个事务有两种方式，一种是显式开始，一种是隐式开始。使用START TRANSACTION或者BEGIN等语句开始一个事务，属于显式开始一个事务。在前面讲过的SQL语句中，每一个SQL语句都

可以作为包含一条命令的事务，这属于隐式开始一个事务。

### 1. 显式开始一个新事务

使用START TRANSACTION或者BEGIN语句可以显式开始一个新的事务。其语法格式如下：

```
START TRANSACTION [事务名]
```

其中，START TRANSACTION表示显式开始一个新事务的关键字；START TRANSACTION关键字的后面跟的是事务的名字。事务名是可选的。

#### 说明

Oracle数据库和Microsoft SQL Server数据库还支持使用BEGIN TRANSACTION语句显式开始一个新事务。

#### 注意

如果想结束该事务可以使用COMMIT或者ROLLBACK语句。其中，COMMIT表示提交事务；ROLLBACK表示回滚事务。

### 2. 隐式开始一个事务

在一个应用程序中，从每一个SQL语句中的第一条语句开始就表示隐式开始了一个新的事务。默认情况下，数据管理系统会将使用SQL语句开始的新事务以自动提交的方式运行。

## 16.2.2 提交事务

提交事务可以有三种方式：显式提交、隐式提交和自动提交。显式提交是通过使用COMMIT语句完成的事务的提交；隐式提交是指通过使用SQL语句就可以完成事务的提交；自动提交指通过设置AUTOCOMMIT命令完成事务的提交。下面就来介绍这3种提交方式。

### 1. 显式提交

显式提交一个事务可以使用COMMIT语句。COMMIT语句表示提交事务并保存对该事务所做的修改。其语法格式如下：

```
COMMIT [事务名]
```

其中，COMMIT 表示提交事务的关键字；COMMIT关键字后面跟的是事务的名字，这个事务的名字是可选的。

#### 例16.1 开始并提交一个事务。

```
START TRANSACTION
INSERT INTO accountTable VALUES ('A',5000);
UPDATE accountTable SET cashValue=cashValue+1000 WHERE accoutUser = 'A';
COMMIT
```

这里使用START TRANSACTION显式开始一个新的事务，在该事务中有两条SQL语句，一条SQL语句是向accountTable表中插入一条记录；另一条SQL语句是修改accountTable表中accoutUser为A的数据记录，这两条SQL语句都被认为在同一个事务中，最后使用COMMIT语句将该事务提交。事务提交之后，数据库会将修改后的记录保存。其他事务也可以查询到该事务提交后新的数据变化。

### 2. 隐式提交

隐式提交是指通过使用SQL语句就可以完成事务的提交。如果执行了CREATE TABLE、CREATE

## 零基础学SQL

INDEX、ALTER TABLE、DROP TABLE、DROP DATABASE、EXIT、QUIT、GRANT、REVOKE等SQL语句，则会隐式地自动提交事务。

### 3. 自动提交

自动提交指通过设置AUTOCOMMIT命令完成事务的提交。如果将AUTOCOMMIT设置为1或者ON，则表示自动提交事务，此时在执行了增加、修改和删除数据的SQL语句之后，数据库管理系统会将该事务自动提交。设置自动提交的方式如下：

```
SET AUTOCOMMIT =1
SET AUTOCOMMIT ON
```

这里提供了两种设置事务自动提交的方式，第一种方式是将AUTOCOMMIT的值设置为1，第二种方式是将AUTOCOMMIT的值设置为ON。

如果要想关闭自动提交方式，可以将AUTOCOMMIT设置为0或者OFF。关闭自动提交的语法规则如下：

```
SET AUTOCOMMIT =0
SET AUTOCOMMIT OFF
```

### 16.2.3 回滚事务

回滚事务是表示当事务执行失败时，数据库将恢复到该事务操作之前的那一个合法状态中，并撤销该事务对数据库所做的包括增加、修改和删除数据在内的所有更新操作。回滚事务可以使用ROLLBACK语句来完成。回滚事务的语法规则如下：

```
ROLLBACK [TO 保存点]
```

其中，ROLLBACK表示用于事务回滚的关键字；TO 保存点是可选的。这里的保存点用来表示回滚部分事务的一种机制。设置保存点后，在执行包括增加、修改和删除数据在内的更新操作时，如果执行过程中没有错误，就继续执行后面的语句；如果执行过程中出现错误，事务就会取消保存点后面的操作，回滚到更新操作之前的保存点。

在数据库中，可以设置保存点。Oracle数据库和Microsoft SQL Server数据库中设置保存点的方法有所不同。在Oracle数据库中，设置保存点的语法规则如下：

```
SAVEPOINT 保存点
```

其中，SAVEPOINT表示设置保存点的关键字；关键字SAVEPOINT后面跟的是保存点的名字。

在Microsoft SQL Server数据库中，设置保存点的语法规则如下：

```
SAVE TRANSACTION 保存点
```

其中，SAVE TRANSACTION表示设置保存点的关键字；关键字SAVE TRANSACTION后面跟的是保存点的名字。

例16.2 回滚事务并为事务设置保存点。

```
START TRANSACTION
INSERT INTO accountTable VALUES ('A',5000);
SAVEPOINT a1;
INSERT INTO accountTable VALUES ('B',5000);
```

```
SAVEPOINT a2;  
INSERT INTO accountTable VALUES ('C',5000);  
SAVEPOINT a3;  
ROLLBACK TO a2;  
COMMIT
```

这段SQL语句中，设置了3个保存点，分别是a1、a2、a3。并使用ROLLBACK语句指定回滚的保存点为a2。即如果执行了ROLLBACK TO a2语句，则事务就会取消保存点a2后面的操作，回滚到更新操作之前的保存点。

## 16.3 事务的并发控制

事务的并发控制是指多个用户同时对同一个数据对象（例如，数据表、视图等）进行更新操作。在执行并发操作的过程中，可能会带来一些问题。这一节就来介绍并发事务的工作流程以及并发事务带来的问题，最后给出了解决事务并发处理中存在问题的方法，并介绍如何在数据库中设置隔离级别与事务访问模式。

### 16.3.1 并发事务的工作流程

在讲解如何对事务进行并发控制之前，有必要先了解一下并发事务的工作流程。为了模拟多个事务的并发操作，需要对同一个数据表建立两个连接，然后让两个用户试图同时修改数据表中的数据。

在模拟事务的并发操作之前，首先需要做一些准备工作。这里首先需要创建一个用于测试并发事务的数据库test\_transaction。创建test\_transaction数据库的SQL语句如下：

```
CREATE DATABASE test_transaction;
```

然后在这个test\_transaction数据库中创建一个数据表accountTable，该数据表用来存储用户账户信息。创建accountTable表的SQL语句如下：

```
CREATE TABLE accountTable(  
  accoutUser VARCHAR(10)PRIMARY KEY,  
  cashValue INT,  
  CHECK(cashValue>=0)  
)
```

在这个accountTable数据表中，包含有两列，列accoutUser表示指定的账户信息；列cashValue表示账户中存储的现金数。这里为了讲解方便，将其定义为整型。其中，列accoutUser定义为主键列，为列cashValue定义一个CHECK约束，让列cashValue中指定的值都大于等于0。

创建完成accountTable表之后，还需要在accountTable数据表中使用INSERT INTO语句增加一条记录。其SQL语句如下：

```
INSERT INTO accountTable  
VALUES ('A',5000)
```

这段SQL语句是向accountTable数据表中增加一条用户账户记录。增加的这条数据记录表示在accountTable表中账户A中有5000元的现金。

上述准备工作完成之后，就可以模拟多个事务的并发操作了。这里以MySQL 5.0数据库为例，模拟两个事务并发操作的工作流程。为了模拟事务的并发操作，需要打开两个MySQL 5.0 Command Line





Client窗口，分别模拟两个不同的用户，然后在这两个窗口中轮流执行一些SQL语句，修改accountTable数据表中的数据，可以通过表16.1来描述两个事务的并发过程。

表16.1 两个并发事务的工作流程

时刻	用户User1	用户User2
时刻1	<pre>SELECT accoutUser,cashValue FROM accountTable; 查询结果: +-----+-----+-----+   accoutUser   cashValue  +-----+-----+-----+   A   5000  </pre>	
时刻2	<pre>SELECT accoutUser,cashValue FROM accountTable; 查询结果: +-----+-----+-----+   accoutUser   cashValue  +-----+-----+-----+   A   5000  </pre>	<pre>SELECT accoutUser,cashValue FROM accountTable; 查询结果: +-----+-----+-----+   accoutUser   cashValue  +-----+-----+-----+   A   5000   处理事务，修改accountTable表: START TRANSACTION UPDATE accountTable SET cashValue=cashValue-1000 WHERE accoutUser = 'A'; 查询结果: +-----+-----+-----+   accoutUser   cashValue  +-----+-----+-----+   A   4000  </pre>
时刻3	<pre>修改accountTable表: START TRANSACTION UPDATE accountTable SET cashValue=cashValue-5000 WHERE accoutUser = 'A';</pre>	
时刻4		COMMIT
时刻5	<pre>余额不足 ROLLBACK</pre>	<pre>SELECT accoutUser,cashValue FROM accountTable; 查询结果: +-----+-----+-----+   accoutUser   cashValue  +-----+-----+-----+   A   4000  </pre>
时刻6	<pre>SELECT accoutUser,cashValue FROM accountTable; 查询结果: +-----+-----+-----+   accoutUser   cashValue  +-----+-----+-----+   A   4000  </pre>	<pre>SELECT accoutUser,cashValue FROM accountTable; 查询结果: +-----+-----+-----+   accoutUser   cashValue  +-----+-----+-----+   A   4000  </pre>

(1) 在时刻1，用户user1使用SELECT语句查询到accountTable表中账户A的余额为5000元，此时由于某种原因，用户user1并没有提取现金。

(2) 在时刻2，用户user2也使用SELECT语句查询到accountTable表中账户A的余额为5000元，并且从账户A中取出了1000元，此时他并没有提交事务。此时用户user1查询到的accountTable表中账户A的余额还是5000元。

(3) 在时刻3，用户user1在查询到accountTable表中账户A的余额为5000元后，希望从账户A中将其全部取出，此时由于user2的事务并没有提交，所以用户user1的这个操作无法继续进行，只能等待用户user2完成他的事务操作。

(4) 在时刻4，用户user2取出1000元后，使用COMMIT命令将该事务提交，此时账户A中的余额为4000元。

(5) 在时刻5，用户user1可以执行他的操作了。当他使用UPDATE语句希望从账户A中将5000元余额全部取出时，却被告知余额不足5000元，不能完成此次操作。此时，由于数据的一致性遭到破坏，用户user1的操作将会被撤销，回滚到他操作之前的状态，而用户user2查询到的账户A中的余额为4000元。

(6) 在时刻6，用户user1和用户user2再查询accountTable表账户A中的余额，发现账户A中余额已经变成了4000元。如果用户user1还希望将账户A中的余额全部取出，就要重新开始一个新的事务去修改账户A中的余额。

### 16.3.2 事务并发处理中存在的问题

从上面的并发事务的工作流程中，读者可能会看出一些问题。确实，事务的并发操作虽然可以提高数据库管理系统对数据处理效率，但同时事务并发处理中也存在一些问题。事务并发处理中主要存在以下3方面的问题。

#### 1. 读脏数据

这里的脏数据是指那些已经更改但是并没有被提交的中间数据。例如，图16.1表示的两个并发事务的工作流程中，账户A中有5000元钱，开始用户user1和用户user2在查询时，都发现账户A中的余额为5000元。用户user2从账户A中取走了1000元，此时账户A中的余额剩下4000元，与此同时，用户user1希望将5000元全部取走，这时会发现账户的余额不足，因此用户user1不得不取消此次操作。也就是说，用户user1读取了并未提交的数据。

#### 2. 不能重复读

这里所说的不能重复读是指同一个事务在多次执行时，由于其他事务对其做的修改或者删除等更新操作，使得每次查询时返回的数据结果都不相同。也就是说，在重复读取数据表中某一个列的数据时，会显示不一致的错误数据。例如，图16.1表示的两个并发事务的工作流程中，在时刻3，用户user2已经将账户A中的余额修改为4000，但是由于没有提交事务，此时，用户user1查询到的账户A中的数据仍然是5000。两个用户此时都查询accountTable数据表中列cashValue中的数据，但是对两个用户却显示出不同的查询结果。当在时刻6，用户A将操作回滚之后，发现此时的账户余额已经变成了4000，这与他在时刻3查询到的账户余额为5000元的数据不同，也就是说，由于其他用户对accountTable数据表中列cashValue中的数据进行了修改操作，使得用户user1两次查询到的结果是不同的。

#### 3. 幻象读

所谓幻象读，是指读取了其他事务中执行完插入或者更新操作后的错误数据。例如，在accountTable

表中目前只有一条数据记录，当用户user1通过指定的SELECT语句查询时，会查询到accountTable表的这一条记录。此时，一个事务向accountTable表中插入了一条数据记录，当用户user1再次查询accountTable表时，会发现在accountTable表中多出了一条记录，会查询到2条数据记录。

### 16.3.3 事务的隔离级别

为了解决事务并发处理中存在的问题，SQL92中定义了4种级别的事务隔离。可以通过使用事务的隔离级别来解决16.3.2节提到的事务并发处理中存在的问题。下面就对这4种隔离级别分别做以介绍。

- ❑ **READ UNCOMMITTED**：未提交读。以图16.1表示的两个并发事务的工作流程为例，如果该例中事务的隔离级别是READ UNCOMMITTED，那么在时刻2中，用户user1查询到的accountTable数据表中列cashValue中的数据就是4000。READ UNCOMMITTED隔离级别下，会隔离UPDATE语句，但不隔离SELECT语句。也就是说，在READ UNCOMMITTED隔离级别下，UPDATE语句会得到正确的执行。它的隔离级别最低。
- ❑ **READ COMMITTED**：提交读。给隔离级别在读取数据时对其加共享锁，可以避免读脏数据。但是在READ COMMITTED隔离级别下，事务在结束前更改可以更改数据，因此不能避免不能重复读或者幻象读。Microsoft SQL Server数据库中，默认的隔离级别是READ COMMITTED。
- ❑ **REPEATABLE READ**：可重复读。该隔离级别会将查询中使用的所有数据锁定，防止其他用户对该数据进行修改操作，可以避免产生不能重复读。但是在REPEATABLE READ级别下允许其他事务插入数据，因此不能避免幻象读。MySQL 5.0数据库中InnoDB数据表的驱动程序使用的是REPEATABLE READ隔离级别。
- ❑ **SERIALIZABLE**：可串行化。该隔离级别在事务提交之前，会锁定整个数据表，防止其他用户对数据进行增加、修改和删除等更新操作，可以避免幻象读。它的隔离级别最高。

**说明** 在Oracle数据库中，不支持READ UNCOMMITTED和REPEATABLE READ这两种隔离级别。

**注意** 不同的隔离级别在解决并发事务时会出现不同的结果。不同的数据库系统也有不同的默认隔离级别。

### 16.3.4 在数据库中设置隔离级别

在数据库中可以使用SQL语句设置隔离级别，可以为一个事务设置隔离级别，也可以为一个会话中的每个事务设置隔离级别。

#### 1. 为一个事务设置隔离级别

在数据库中，可以使用SQL语句设置一个事务的隔离级别。使用SQL语句设置事务的隔离级别的语法规则如下：

```
SET TRANSACTION ISOLATION LEVEL 隔离级别
```

其中，SET TRANSACTION表示为当前的事务设置特性；ISOLATION LEVEL表示为事务设置隔离级别的关键字；在SET TRANSACTION ISOLATION LEVEL关键字后，需要指定隔离级别。

**例16.3** 设置事务的隔离级别为可重复读。

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

这段SQL语句是将事务的隔离级别设置为可重复读。其中REPEATABLE READ表示隔离级别为可重复读。

## 2. 为一个会话中的事务设置隔离级别

在数据库中，除了可以使用SQL语句为一个事务设置隔离级别，还可以为一个会话中的事务设置隔离级别。如果要在MySQL数据库中为一个会话中的事务设置隔离级别，可以使用下面的SQL语句。

```
SET SESSION TRANSACTION ISOLATION LEVEL
[ READ UNCOMMITTED
| READ COMMITTED
| REPEATABLE READ
| SERIALIZABLE ]
```

其中，SET SESSION TRANSACTION表示为当前会话中的事务设置特性；ISOLATION LEVEL表示为事务设置隔离级别的关键字；READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ、SERIALIZABLE表示隔离级别。

如果要在Oracle数据库或者是Microsoft SQL Server数据库中，为一个会话中的事务设置隔离级别，可以使用下面的SQL语句。

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
[ READ UNCOMMITTED
| READ COMMITTED
| REPEATABLE READ
| SERIALIZABLE ]
```

其中，SET SESSION CHARACTERISTICS表示为当前会话中的事务设置特性；ISOLATION LEVEL表示为事务设置隔离级别的关键字；READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ、SERIALIZABLE表示隔离级别。

**注意** 在使用SQL语句设置事务的访问级别时，SET TRANSACTION 必须是事务处理的第一条语句。

### 16.3.5 设置事务访问模式

在Oracle数据库中，还可以为事务设置访问模式。事务的访问模式包括只读模式和读写模式。其中，只读模式只允许是在执行查询操作，而不允许事务执行INSERT、UPDATE、DELETE、GRANT、REVOKE等操作。设置事务访问模式的语法规则如下：

```
SET TRANSACTION
[ READ ONLY | READ WRITE ]
```

其中，SET TRANSACTION表示为当前的事务设置特性；READ ONLY和READ WRITE表示事务的访问模式，READ ONLY表示只读模式，READ WRITE表示读写模式。

**例16.4** 设置事务为只读模式。

```
SET TRANSACTION
READ ONLY
```

**注意** 在使用SQL语句设置事务的访问模式时，SET TRANSACTION READ ONLY必须是事务处理的第一条语句。

**说明** Microsoft SQL Server数据库和MySQL数据库不支持使用SET TRANSACTION设置事务访问模式。

## 16.4 小结

本章主要讲解了事务及事务控制和管理的相关内容。通过本章的学习，读者需要掌握事务需要满足的ACID属性以及如何使用START TRANSACTION、COMMIT、ROLLBACK等语句控制事务。另外还需要了解并发事务的工作流程以及并发事务中存在的问题。SQL92标准中事务的4种隔离级别，按照隔离级别从低到高依次为READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ和SERIALIZABLE。通过使用SET TRANSACTION 可以设置访问级别和事务的访问模式。其中，事务的访问模式包括只读模式和读写模式两种。



## 第六篇 PL/SQL

---

### 第17章 PL/SQL概述

PL/SQL (Procedural Language/SQL) 是Oracle对标准数据库SQL语言的扩展。在PL/SQL中，不仅可以使用SQL语句，而且还可以在PL/SQL控制语句中定义变量和常量、使用条件分支和循环控制语句、可以进行异常处理、可以使用游标进行数据操作等。在任何运行Oracle的平台中，都可以使用PL/SQL进行应用程序的开发。这一节就来介绍PL/SQL的一些基本内容。

本章重点：

- ☐ 什么是PL/SQL
- ☐ 使用PL/SQL的原因
- ☐ 介绍一个简单的PL/SQL例子
- ☐ PL/SQL开发工具简介
- ☐ PL/SQL的编写规范

#### 17.1 PL/SQL介绍

PL/SQL作为一种可以运行在任何Oracle平台中的较为复杂的程序设计语言，在实际的开发应用中得到了广泛的应用。这一节将对PL/SQL做一个概括性的介绍，并结合一个实例讲解为什么要在程序应用中使用PL/SQL。

##### 17.1.1 什么是PL/SQL

PL/SQL是Oracle对标准数据库SQL语言的扩展，它可以运行在任何的Oracle开发环境中。Oracle公司已经将PL/SQL语言集成到Oracle的服务器中，也可以在由其他的第三方提供的工具中使用PL/SQL。

PL/SQL语言是一种比较复杂的数据库程序设计语言，它将第四代语言（如SQL）的灵活性与第三代语言（如C、COBOL、C++、Java等）的过程性结构结合在一起，很好地融合了SQL语言中强大的数据库操作性能以及程序设计语言中的过程结构。因此，PL/SQL语言的功能强大，可以用它来解决数据库应用中复杂的应用程序。目前，很多的开发人员和数据库管理人员（DBA）在实际应用中都开始使用PL/SQL语言。

PL/SQL语言中的PL的英文全称是Procedural Language，顾名思义，作为对SQL语言的扩展，PL/SQL语言增加了许多程序设计语言中的过程结构。主要包括以下几个方面的扩展。

- 变量。例如，变量声明、变量初始化等。
- 数据类型。例如，标量类型、复合类型、引用类型、对象类型、用户自定义类型等。
- 结构控制语句。例如，条件语句IF-ELSE-THEN、CASE和循环语句LOOP、WHILE等。
- 过程和函数。在数据库中以编译好的形式存储，供其他的语句块调用。
- 异常处理。用来处理程序运行时出现的错误，将程序逻辑处理代码与错误处理代码分离。

### 17.1.2 为什么要使用PL/SQL

前面介绍SQL语言时，已经知道，使用SQL可以很灵活地操作数据库，实现数据表（或者视图）的创建、数据的查询、增加、修改、删除、表结构的修改、权限的授予与回收等操作，那么为什么要使用PL/SQL语言呢？

在实际的应用中，有些时候需要处理的业务逻辑可能会比较复杂。现在考虑这样一个问题，一所学校要对某一名教师的职称进行修改，如果这个教师不存在，就向教师信息表中插入一条该教师的记录；如果这个教师存在，就在教师信息表中修改这名教师的职称信息。对于这个问题，如果仅仅使用SQL语言是无法来完成的，因为在SQL语言中，并没有提供用于判断的语句命令。

那么使用PL/SQL可以完成上面的这个业务逻辑吗？答案是肯定的。因为PL/SQL语言作为对SQL语言的扩展，提供了像条件判断、循环判断这样的结构控制语句，因此类似于上面的问题（甚至比上面的问题还复杂的问题）可以很容易地用PL/SQL语言来解决。其实现的主要代码如下所示。

```
DECLARE
    v_ teaID VARCHAR2(15) := 't152303';
    v_ teaName VARCHAR2(10) := '王杰';
    v_ age NUMBER(2) := 45;
    v_ dept VARCHAR2 (20) := '计算机系';
    v_ profession VARCHAR2 (10) := '教授';
    v_ salsry NUMBER(6,2) := 5000;
BEGIN
    UPDATE t_ teacher
    SET profession = v_ profession
    WHERE teaID = v_ teaID;
    IF SQL%NOTFOUND THEN
        INSERT INTO T_student
        VALUES(v_ teaID, v_ teaName, v_ age, v_ dept, v_ profession, v_ salsry);
    END IF;
END;
```

从这段PL/SQL的代码可以看到，这里将两个SQL语句绑定到一个PL/SQL块中，作为一个独立的单元发送给服务器执行，而且在这段PL/SQL的代码中，还可以包括变量的声明以及IF这样的条件判断语句。（有关这个例子的讲解可以参看20.3.2小节）。

读者可能会提出这样的疑问，对于上面的这个问题，不使用PL/SQL语言，通过程序设计语言和SQL语言相结合的方式也可以很容易地解决这个问题。以Java语言为例，可以通过JDBC与数据库服务器进行连接，然后根据Java语言中的if语句对查询结果进行判断，如果教师记录不存在，就通过INSERT INTO语句增加这个教师的信息，如果这个教师存在，就通过UPDATE语句修改该名教师的职

称信息。

但是这种方式也会带来一个问题，因为在实际应用中，数据库服务器一般都会在一个独立的机器上，而编写程序代码实现业务逻辑的软件开发人员会在其他机器上来实现自己的应用程序，也就是说数据库服务器与应用程序服务器会在不同的机器上。在这样的一个网络环境中实现程序设计语言与数据操作之间的交互，对于发送到数据库服务器的每一条SQL语句，Oracle必须在同一时间对每一个这样独立的SQL语句进行处理，会占用很多服务器的时间，相应地就会加大网络的负载，降低了程序的执行效率。而PL/SQL语言是将多条SQL语句以语句块的形式绑定在一起，将其作为一个独立的单元直接发送给服务器，这就降低了网络负载，提高了程序的执行效率。

当然，这里也需要说明的一点是，PL/SQL语言是集成在Oracle服务器中，因此在Oracle数据库中使用PL/SQL语言执行相应的数据库操作，其执行速度会更快，但是也正是由于PL/SQL语言是集成在Oracle服务器中的，所以很难将它从一种数据库移植到另一种数据库中。例如，从Oracle数据库移植到MySQL数据库、从Oracle数据库移植到Microsoft SQL Server数据库等）。

## 17.2 一个PL/SQL的例子

为了让读者能够了解PL/SQL语句块的基本组成，这里给出一个使用PL/SQL语言完成查询学生信息表（t\_student）中指定学生编号对应的学生姓名的例子。在该例中，如果输入的学生编号在学生信息表中存在，则将该学生的信息输出，如果输入的学生编号在学生信息表中不存在，则显示“学生编号不存在”的错误信息，并将该信息写入到日志表（t\_log）中。其代码如下：

```
DECLARE
    v_stuName VARCHAR (10);           -- 学生姓名
BEGIN
    /*
    *通过学生编号从学生信息表中
    *取得对应的学生姓名信息
    */
    SELECT v_stuName
        INTO v_stuName
    FROM t_student
    WHERE stuID = &stuID;
    DBMS_OUTPUT.PUT_LINE ('学生姓名为' || v_stuName);    -- 显示学生姓名
EXCEPTION
    WHEN NO_DATA_FOUND THEN           -- 异常处理
        DBMS_OUTPUT.PUT_LINE ('学生编号不存在')         -- 显示错误信息
        INSERT INTO t_log (information,sysdate,user)     -- 将错误信息写入日志文件
        VALUES('stuID does not exist!',SYSDATE, 'admin');
END;
```

可以看出，在这个PL/SQL语句块主要包括了三个部分：DECLARE、BEGIN和EXCEPTION。其中，DECLARE是定义部分，用于定义变量、常量、游标、数据类型等；BEGIN是执行部分，通过执行PL/SQL和SQL语句来实现业务逻辑；EXCEPTION是异常处理部分，用于处理程序运行时可能出现的错误。其中，t\_log表是一个用来记录错误信息的日志表。这个表中包含三个字段，第一个字段用来记录错误信息，第二个字段记录产生错误信息的时间，第三个字段记录操作者。另外，使用“--”和

“/\*\*/”的部分是PL/SQL语言的注释部分。

在DECLARE、BEGIN和EXCEPTION这三个部分中，只有BEGIN部分是必需的，其他两个部分都是可选的。这里只是使用PL/SQL语言实现一个非常简单功能的例子，使用PL/SQL还可以用来编写存储过程、函数、包、触发器等实现更为复杂的应用。有关PL/SQL语言的这些内容，将在以后的几章陆续地介绍。

## 17.3 PL/SQL编写规范

PL/SQL语言是作为一种高级的数据库程序设计语言，在编写时也应该遵循一定的编写规范。良好的程序编写规范不仅可以让其程序更容易阅读和理解，而且也有利于以后程序代码的维护。这一节就来介绍编写PL/SQL语言中常用的一些编程规范。

### 17.3.1 代码注释

在实际应用中，无论用任何一种程序设计语言编写的代码都不能没有注释。在PL/SQL的代码中，一般需要在三个地方为其添加注释。

- ☐ 每一个PL/SQL语句块或者过程的开始部分。用于说明这个PL/SQL语句块或者过程是用来干什么的。
- ☐ 每一个变量声明的后面。用于说明该变量所表示的意义。
- ☐ PL/SQL语句块或者过程中的主要业务逻辑的处理部分。用于说明该段处理程序的作用以及可能产生的结果。

### 17.3.2 标识符命名

作为程序设计语言，在实际应用中都会使用标识符命名变量、常量等，在PL/SQL语言中，标识符名称字符长度最大不能超过30个字符，并且需要以字母开头。除此之外，对于标识符命名时最好遵循以下的规则。

- ☐ 以v\_开头定义变量。例如，v\_stuID、v\_stuName等。
- ☐ 以c\_开头定义常量。例如，c\_constantNumber。
- ☐ 以e\_开头定义用户自定义异常。例如，e\_illegalValue。
- ☐ 以t\_开头定义用户自定义类型变量。例如，t\_userDefinedType。
- ☐ 以\_cursor做前缀定义游标。例如，cursor\_student。
- ☐ 以\_record做后缀定义一个记录。例如，stu\_record。

要为变量起有意义的名字。也就是说，通过变量名可以让阅读程序的人知道这个变量是用来做什么的。例如，如果要定义一个学生编号的变量，那么v\_stuID的变量名就要比a的变量名更有意义。

### 17.3.3 字母大小写

虽然PL/SQL语言不区分大小写，但是在程序中适当区分字母的大小写还是有利于程序的可读性。因此，在使用PL/SQL语言时字母的大小写可以遵循以下的规则。

- ☐ PL/SQL语言关键字需要大写。例如，DECLAR、BEGIN等。
- ☐ SQL语言中的关键字需要大写。例如，SELECT、WHERE、INSERT、UPDATE等。

- ❑ SQL语言中内置函数需要大写。例如，ROUND()、TO\_DATE()、LENGTH()等。
- ❑ 数据类型需要大写。例如，INT、VARCHAR2等。
- ❑ 数据库对象使用小写字母。例如，teacher、student等。
- ❑ 变量和参数使用小写字母。例如，v\_stuID、stu1\_record等。

### 17.3.4 代码缩进格式

为了使程序更容易阅读和理解，一般在一段代码中都会使用制表符（“Tab”键）对代码进行缩进。例如，下面这个例子。

```
DECLARE
    v_digital INT:= 1;
    v_sum INT:= 0;
BEGIN
    WHILE v_digital<10 LOOP
        v_sum = v_sum+ v_digital;
        v_digital:= v_digital+1;
    END LOOP;
END;
```

这段PL/SQL代码中，WHILE...LOOP是一个循环语句，通过循环语句来计算1到10的加和，并将结果赋值给变量v\_sum。


在PL/SQL语言中，如果使用条件语句和循环语句，更要注意代码的缩进，否则会影响对程序的阅读和理解。

## 17.4 PL/SQL开发工具简介

Oracle公司和许多第三方供应商都提供了许多不同的PL/SQL开发工具。其中主要的PL/SQL开发工具包括SQL\*Plus、TOAD、Navicat和PL/SQL Developer。其中，SQL\*Plus是和Oracle服务器集成在一起的，是Oracle安装的一部分。它是最简单的PL/SQL开发工具。在提示行中输入PL/SQL语句块会直接发送给数据库，执行后的结果会返回到屏幕上。本节主要介绍除SQL\*Plus以外的其他三种PL/SQL开发工具。

### 17.4.1 TOAD

TOAD是一个强大、低开销的PL/SQL开发工具，它只需占用很少的内存，可以满足管理、开发、性能调优等要求，是Oracle开发者常用的工具之一。其安装界面如图17.1所示。

安装完成后，在桌面上会出现一个图标，双击该图标，可以进入到TOAD的开发环境中，在第一次进入到开发环境之前，首先要进行登录连接，在登录连接的对话框中需要输入用户名、密码、数据库以及数据库连接的信息。其登录连接的对话框如图17.2所示。

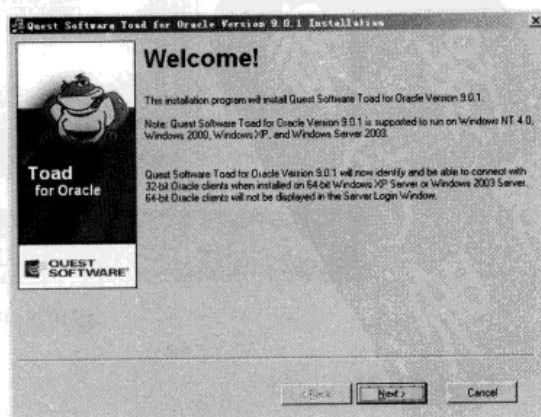


图17.1 TOAD开发工具安装界面



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

零基础学SQL

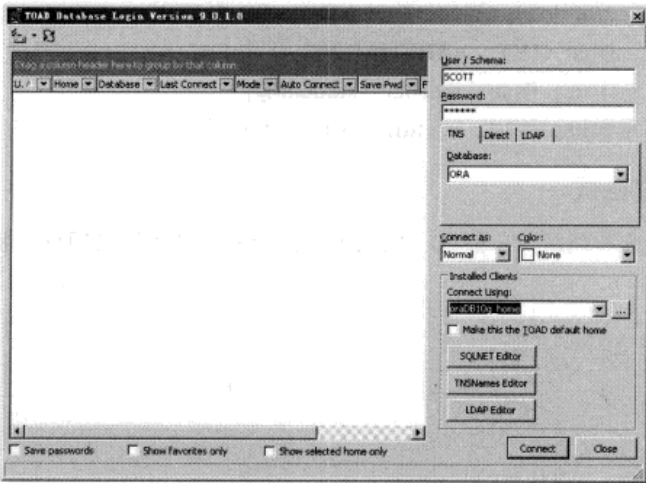


图17.2 TOAD登录连接的对话框

在输入完数据库的连接信息之后，单击“Connect”按钮，会进入到相应的PL/SQL语句块编辑界面。在这个编辑界面中可以很方便地编辑PL/SQL语句块，还可以对其进行代码格式化等操作。在编辑界面的下方有一个显示结果的区域，PL/SQL语句块的执行结果会在显示结果区中显示，如图17.3所示。

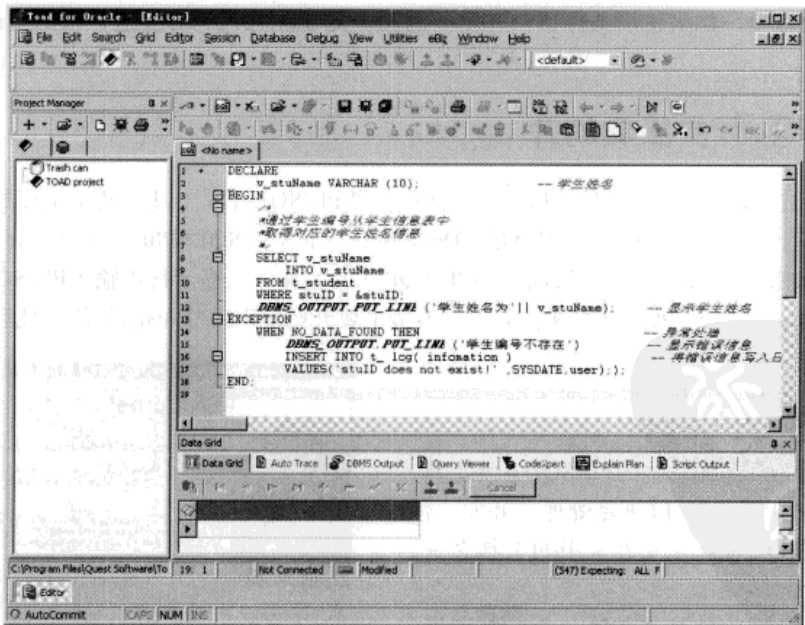



图17.3 PL/SQL编辑界面

17.4.2 Navicat

Navicat是由Quest Software公司出品的数据库管理工具，支持PL/SQL的调试，可以自动格式化

PL/SQL 语句，还可以进行数据同步和批处理管理、支持向导的导入和导出、日程备份等功能。其安装界面如图 17.4 所示。

安装完成后，在桌面上会出现一个  图标，双击该图标，可以进入到 Navicat 的开发环境中，如图 17.5 所示。

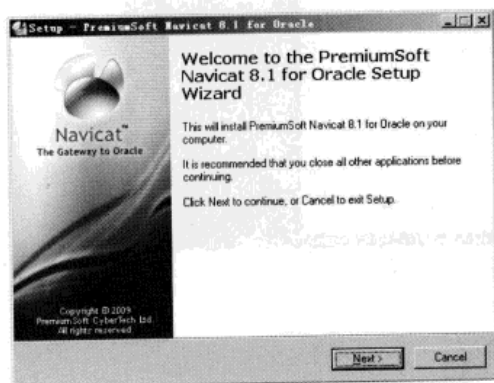


图 17.4 Navicat 开发工具安装界面

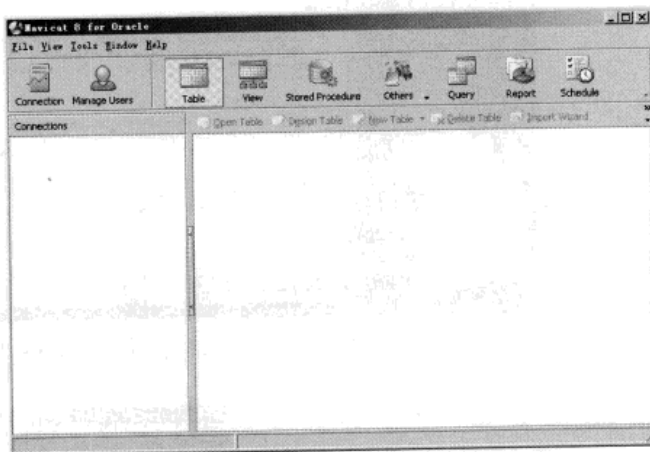



图 17.5 Navicat 的开发环境界面

在第一次进入到开发环境之前，同样也需要进行登录连接。可以单击图 17.5 中的“Connection”图标，对其进行登录连接的配置，如图 17.6 所示。

在输入完数据库的连接信息之后，单击“Test Connection”按钮，就可以完成对数据库的连接操作进入到相应的 PL/SQL 语句块编辑界面。

### 17.4.3 PL/SQL Developer

PL/SQL Developer 是一个专门用于 Oracle 数据库 PL/SQL 的程序编写与调试程序的集成开发环境，其安装使用非常简单，可以方便地对 Oracle 数据库 PL/SQL 的程序进行编辑、编译、测试、调试以及性能优化，是 Oracle 开发者常用的工具之一。其安装界面如图 17.7 所示。

安装完成后，在桌面上会出现一个  图标，双击该图标，可以进入到 PL/SQL Developer 的开发环境中。在第一次进入到开发环境之前，同样也需要进行登录连接。在登录连接的对话框中需要输入用户名、密码以及数据库连接的信息。其登录连接的对话框如图 17.8 所示。

在输入完数据库的连接信息之后，单击“OK”按钮，会进入到相应的 PL/SQL Developer 开发界面。PL/SQL Developer 开发界面如图 17.9 所示。

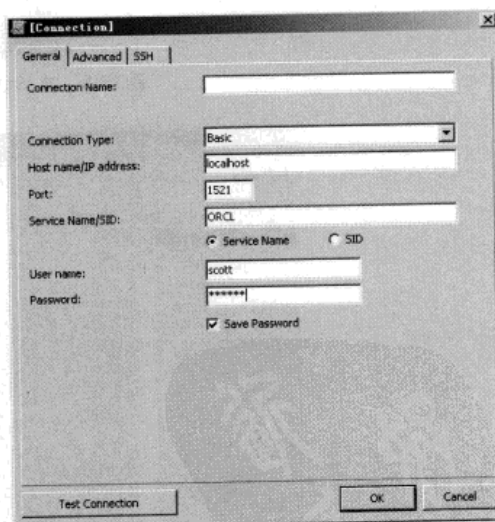


图 17.6 Navicat 登录连接的对话框

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

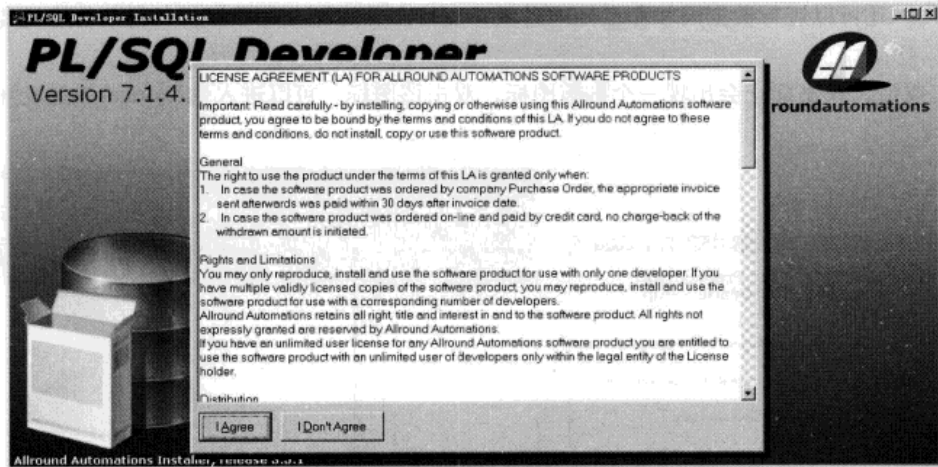


图17.7 PL/SQL Developer开发工具安装界面

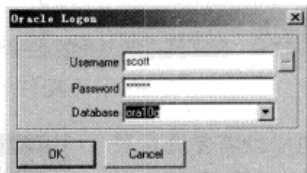


图17.8 PL/SQL Developer登录连接的对话框

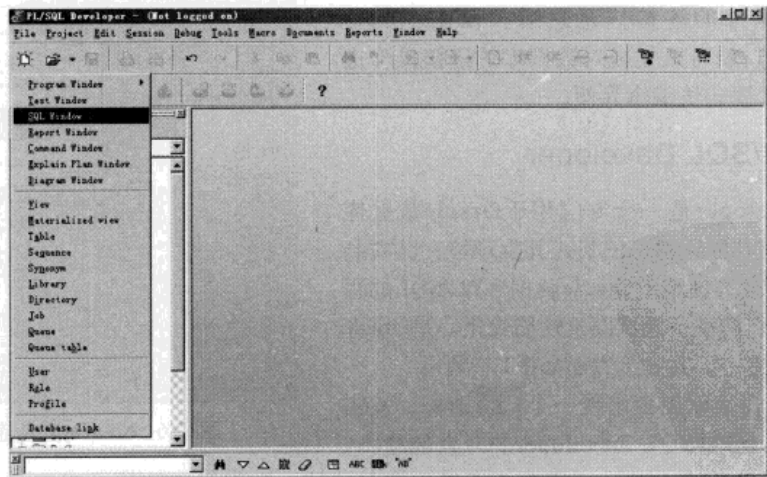


图17.9 PL/SQL Developer开发界面

如果想在PL/SQL Developer开发界面中编辑PL/SQL 语句块，可以选择图17.9列表中的“SQL Window”选项。此时会进入到SQL Window界面，可以在这里编写PL/SQL 语句块，如图17.10所示。

如果想执行SQL Window界面中的PL/SQL 语句，可以按F8键，则PL/SQL Developer会自动执行该窗口中所有的PL/SQL语句。选中SQL Window界面中“Output”选项卡可以查看到使用DBMS\_

OUTPUT.PUT\_LINE的结果。

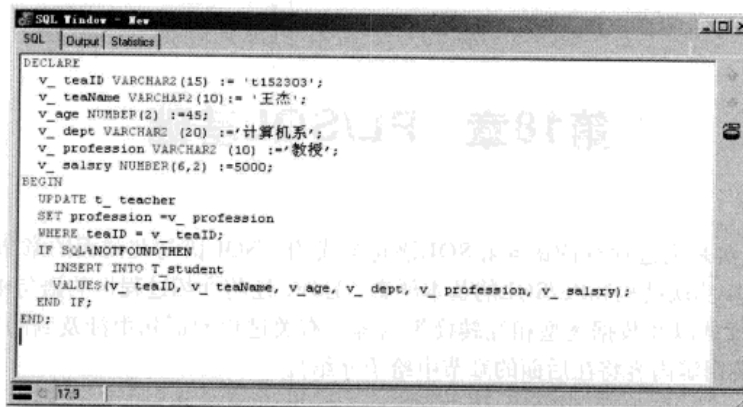


图17.10 SQL Window界面编写PL/SQL 语句块

## 17.5 小结

虽然，这里给出的程序编写规范读者并不一定要完全遵循，但是良好的程序编写规范不仅可以让程序更容易阅读和理解，而且也有利于以后程序代码的维护。本章的最后介绍了PL/SQL编程常用的三种开发工具TOAD、Navicat和PL/SQL Developer。

## 第18章 PL/SQL基础

PL/SQL语句其实是由过程性的语句和SQL语句组成的。SQL语句在前面的章节中已经介绍过了，本章就来介绍PL/SQL的块结构和PL/SQL的基本要素。另外，还将介绍过程性的语句中涉及的变量声明、数据类型、变量作用域以及数据类型相互转换等内容（有关过程性语句中涉及到的其他问题，例如控制结构语句、过程调用等内容将在后面的章节中给予介绍）。

本章重点：

- ☐ PL/SQL中的块结构
- ☐ PL/SQL中的基本要素
- ☐ 变量的声明和初始化
- ☐ PL/SQL中的数据类型
- ☐ 标量变量和复合变量的定义和使用
- ☐ 用户自定义变量的使用
- ☐ 变量的作用域
- ☐ PL/SQL中数据类型之间的转换

### 18.1 PL/SQL的块结构

语句块是PL/SQL语句块的基本单元。PL/SQL语句块就是由语句块构成的。语句块之间可以相互嵌套。这一节就来介绍PL/SQL语句块中几种常用的块结构，包括基本语句块、匿名语句块、命名语句块、子程序、包和触发器。

#### 18.1.1 基本语句块

PL/SQL语言中的基本语句块包含三个部分：定义部分、执行部分和异常处理部分。在这三个部分中，只有执行部分是必需的，而定义部分和异常处理部分是可选的。例如，下面这个例子。

```
DECLARE
/*
*定义部分
*定义变量、常量、游标、数据类型
*/
v_stuName VARCHAR2(10);           -- 学生姓名
BEGIN
/*
*执行部分
*处理业务逻辑，实现应用程序功能
*/
```



```
SELECT v_stuName
      INTO v_stuName
FROM t_student
WHERE stuID = &stuID;          -- 根据学生编号取得对应的学生姓名
DBMS_OUTPUT.PUT_LINE ('学生姓名为' || v_stuName); -- 输出显示学生姓名
EXCEPTION
/*
*异常处理部分
*处理程序运行时可能出现的错误
*/
WHEN NO_DATA_FOUND THEN      -- 异常处理
    DBMS_OUTPUT.PUT_LINE ('学生编号不存在') -- 输出显示错误信息
    INSERT INTO t_log         -- 将错误信息写入日志文件
    VALUES('stuID does not exist!', SYSDATE,user);
END;
```

**注意** 在使用PL/SQL定义基本语句块时，关键字DECLARE、BEGIN和EXCEPTION的后面没有分号，而在表示程序结束的关键字END后面需要加上一个分号。

其中，DECLARE是定义部分，用于定义变量、常量、游标、数据类型等；BEGIN是执行部分，通过执行PL/SQL和SQL语句来实现业务逻辑；EXCEPTION是异常处理部分，用于处理程序运行时可能出现的错误。如果在程序执行的过程中没有错误，则EXCEPTION部分后面的代码不会被执行。

程序中的&stuID 表示SQL\*plus中的替代变量，在执行该段程序时，会要求用户输入一个学生编号，然后系统会根据这个学生编号从学生信息表中查找对应的学生姓名信息；DBMS\_OUTPUT是Oracle数据库提供的系统包，PUT\_LINE是其中包含的一个过程，其作用是输出显示相应的字符串信息。

**注意** 如果在SQL\*plus中使用DBMS\_OUTPUT.PUT\_LINE将参数在屏幕上输出，需要将SQL\*plus的环境变量setveroutput设置为on。

这里，DECLARE和EXCEPTION两个部分都是可选的，在使用PL/SQL编写程序时，这两部分可以没有，但是执行部分BEGIN是必需的。例如，下面这个例子中，就只包含BEGIN语句块，而不包含其他两个部分。

```
BEGIN
    DBMS_OUTPUT.PUT_LINE ('BEGIN'); -- 输出显示内容
END;
```

这段PL/SQL语句块中，只包含执行部分BEGIN的语句块。这里是使用Oracle数据库提供的系统包DBMS\_OUTPUT输出显示一个字符串信息。

通过上面的介绍，读者应该了解了PL/SQL基本语句块的结构，下面将PL/SQL基本语句块的结构总结如下。

### 1. 完整的基本语句块架构

PL/SQL语句块中完整的基本语句块包括定义部分、执行部分和异常处理部分。其中，以关键字DECLARE开头的表示定义部分；以关键字BEGIN开头的表示执行部分；以关键字EXCEPTION开头的表示异常处理部分；最后以关键字END结束。关键字END后面需要有一个分号用来结束该语句块的定

义。

```
DECLARE
    /* 定义变量、常量、数据类型、游标等*/
BEGIN
    /*执行代码*/
EXCEPTION
    /*异常处理代码*/
END;
```

## 2. 包含定义部分和执行部分的语句块架构

包含定义部分和执行部分的语句块架构中，以关键字DECLARE开头的表示定义部分，以关键字BEGIN开头的表示执行部分，最后以关键字END结束。关键字END后面需要有一个分号用来结束该语句块的定义。

```
DECLARE
    /* 定义变量、常量、数据类型、游标等*/
BEGIN
    /*执行代码*/
END;
```

## 3. 包含执行部分和异常处理部分的语句块架构

包含定义部分和执行部分的语句块架构中，以关键字BEGIN开头的表示执行部分，以关键字EXCEPTION开头的表示异常处理部分；最后以关键字END结束。关键字END后面需要有一个分号用来结束该语句块的定义。

```
BEGIN
    /*执行代码*/
EXCEPTION
    /*异常处理代码*/
END;
```

## 18.1.2 匿名语句块

所谓匿名语句块，顾名思义，就是没有名字的语句块。匿名语句块只能执行一次，一般都是动态生成的，它既可以嵌入到像C/C++这样的程序中，也可以由客户端程序产生，调用服务器中的一个子程序。例如，下面这个例子中，是使用匿名语句块向学生信息表中插入一条数据，然后再将插入到学生信息表中的这条数据显示出来。

```
DECLARE
    v_ stuID VARCHAR2(15) := 's145203';
    v_ stuName VARCHAR2(10) := '杜玉';
    v_ age NUMBER(2) := 22;
    v_ sex VARCHAR2 (20) := '女';
    v_ birth DATE := '19870513';
BEGIN
    /*插入学生信息*/
    INSERT INTO t_student
```

```
VALUES(v_ stuID, v_ stuName, v_ age, v_ sex, v_ birth);
/*根据学生编号查询学生姓名*/
SELECT stuName
  INTO v_ stuName
 FROM t_ student
 WHERE stuID = v_ stuID;
DBMS_OUTPUT.PUT_LINE ('学生姓名为' || v_ stuName);    -- 显示学生姓名
END;
```

这段PL/SQL语句会将插入到学生信息表中的学生编号为s145203的学生姓名显示到屏幕上，可以看到这段PL/SQL语句没有名字，而是直接以关键字DECLARE开头，所以这是一个匿名语句块。

### 18.1.3 命名语句块

所谓命名语句块，顾名思义，就是给予特定名称的PL/SQL 语句块。它与匿名语句块用法基本相同，也只能执行一次，一般都是动态生成的。

如果要为匿名语句块加入相应的名字，可以使用“<<>>”对其进行标记。例如，这里还是以18.1.2小节中的那个例子为例，为匿名语句块加入相应的标号。

```
<<insert_student>>
DECLARE
  v_ stuID VARCHAR2(15) := 's145203';
  v_ stuName VARCHAR2(10) := '杜玉';
  v_ age NUMBER(2) := 22;
  v_ sex VARCHAR2 (20) := '女';
  v_ birth DATE := '19870513';
BEGIN
  /*
   *插入学生信息
   */
  INSERT INTO t_ student
    VALUES(v_ stuID, v_ stuName, v_ age, v_ sex, v_ birth);
  /*
   *根据学生编号查询学生姓名
   */
  SELECT stuName
    INTO v_ stuName
  FROM t_ student
  WHERE stuID = v_ stuID;
  DBMS_OUTPUT.PUT_LINE ('学生姓名为' || v_ stuName);    -- 显示学生姓名
END;
```

这段PL/SQL语句也会将插入到学生信息表中的学生编号为s145203的学生姓名显示到屏幕上，但是在关键字DECLARE的前面为其加上了一个标号，它也就成为一个命名语句块了。在关键字END的后面也可以添加<<insert\_student>>标记，不过它是可选的。

在PL/SQL的程序设计中，如果语句块之间出现嵌套，这种使用标号的命名语句块可以对嵌套的语句块加以区分。

```
<<outer >>
```

```
DECLARE
BEGIN
    <<inner >>
    BEGIN
        /*内层语句块执行代码*/
    END <<inner >>;
    /*外层语句块执行代码*/
END <<outer >>;
```

在这段PL/SQL语句块中，一共有两个可执行部分的代码。一个是外层的主语句块的BEGIN部分，一个是内层子语句块的BEGIN部分。这里使用<<>>”对这两个部分的语句块进行标记，其中，主语句块使用<<outer >>进行标记，子语句块使用<<inner >>进行标记。

#### 18.1.4 子程序

子程序包括过程、函数。子程序既可以应用在客户端，也可以应用在服务器端。子程序可以通过调用显式地执行。子程序如果生成，就不可更改。PL/SQL的子程序可以在应用程序中多次运行，通过使用子程序，可以提高程序的执行效率，并有利于客户端程序的开发和维护。

##### 1. 过程

在PL/SQL中，可以使用关键字CREATE [OR REPLACE] PROCEDURE创建一个存储过程。其中，关键字OR REPLACE是可选的。如果想把18.1.2节中的例子变成一个存储过程，可以使用下面的代码来实现。

```
CREATE OR REPLACE PROCEDURE insert_student
AS
    v_ stuID VARCHAR2(15) := 's145203';
    v_ stuName VARCHAR2(10):= '杜玉';
    v_ age NUMBER(2) :=22;
    v_ sex VARCHAR2 (20) :='女';
    v_ birth DATE :='19870513';
BEGIN
    /*
    *插入学生信息
    */
    INSERT INTO t_student
        VALUES(v_ stuID, v_ stuName, v_ age, v_ sex, v_ birth);
    /*
    *根据学生编号查询学生姓名
    */
    SELECT stuName
        INTO v_ stuName
    FROM t_student
    WHERE stuID = v_ stuID;
END insert_student;
```

这段代码是通过使用CREATE OR REPLACE PROCEDURE关键字创建了一个存储过程。其中，insert\_student是存储过程的名字。当这个存储过程创建完成之后，就可以使用匿名语句块调用这个存储



过程。其调用方法如下：

```
BEGIN
    insert_student;
END;
```

这里的匿名语句块只包含一个可执行部分BEGIN。其中，insert\_student就是存储过程的名字。最后以关键字END结束。关键字END后面需要有一个分号用来结束该语句块的定义。

当然，在创建过程的时候，还可以使用参数。参数既可以是输入参数，也可以是输出参数，还可以是输入输出参数。其中，输入参数在Oracle数据库中用IN来定义，当使用匿名语句块调用该过程时，会将该参数传递到调用的过程中；输出参数在Oracle数据库中用OUT来定义，该参数的数据可以传递给调用程序；输入输出参数在Oracle数据库中用INOUT来定义，表示该参数既可以作为输入参数，也可以作为输出参数使用。如果存储过程中，参数没有指定是什么类型，则其参数默认模式为IN。例如下面这个例子。

```
CREATE OR REPLACE PROCEDURE insert_student(
    p_stuID VARCHAR2(15),
    p_stuName VARCHAR2(10),
    p_age NUMBER(2),
    p_sex VARCHAR2(20),
    p_birth DATE)
AS
BEGIN
    /*
    *插入学生信息
    */
    INSERT INTO t_student
        VALUES(p_stuID, p_stuName, p_age, p_sex, p_birth);
    /*
    *根据学生编号查询学生姓名
    */
    SELECT stuName
        INTO v_stuName
    FROM t_student
    WHERE stuID = v_stuID;
END insert_student;
```

这里也是使用CREATE OR REPLACE PROCEDURE关键字创建了一个存储过程。在这个名为insert\_student的存储过程中，包括了五个输入参数：p\_stuID、p\_stuName、p\_age、p\_sex和p\_birth。如果使用匿名语句块调用这个存储过程，可以使用下面的代码完成。

```
BEGIN
    insert_student('s145203', '杜玉', 22, '女', '19870513')
END;
```

在这个匿名语句块中，调用了名为insert\_student的存储过程，并向这个存储过程传递了五个参数。当调用这个存储过程时，这些参数就会被传递到过程中。有关存储过程的详细讲解可以参看第22章。

## 2. 函数

在PL/SQL中，可以使用关键字CREATE [OR REPLACE] FUNCTION 定义一个函数。其中，关键



字OR REPLACE是可选的。下面来看一个使用函数的例子。

```
CREATE OR REPLACE FUNCTION insert_student(  
    p_stuID VARCHAR2(15),  
    p_stuName VARCHAR2(10),  
    p_age NUMBER(2),  
    p_sex VARCHAR2(20),  
    p_birth DATE)  
RETURN VARCHAR2  
AS  
BEGIN  
    /*  
    *插入学生信息  
    */  
    INSERT INTO t_student  
        VALUES(p_stuID, p_stuName, p_age, p_sex, p_birth);  
    /*  
    *根据学生编号查询学生姓名  
    */  
    SELECT stuName  
        INTO v_stuName  
    FROM t_student  
    WHERE stuID = v_stuID;  
    RETURN v_stuName;  
END insert_student;
```

从这段PL/SQL代码中可以看到，定义一个函数和定义一个存储过程的使用方法基本是相同的。函数与存储过程一样，都有自己的定义部分、执行部分、异常处理部分，定义部分和异常处理部分同样也是可选的；都可以带有参数，函数中的参数也包括三种模式，既可以是输入参数，也可以是输出参数，还可以是输入输出参数；它们也都可以存储在数据库中。

但是，函数和存储过程也是有所区别的，从关键字中可以看到，函数相对于存储过程来说，有两个关键字是不同的，一个是FUNCTION，还有一个是RETURN。关键字FUNCTION用来表示定义的是一个函数，关键字RETURN则表示该函数的返回值。当定义一个函数时，必须要使用关键字RETURN定义函数的返回值，而在BEGIN执行部分中，也需要使用RETURN关键字指定函数要返回的值。这里定义的insert\_student函数中关键字RETURN后面的VARCHAR2表示函数需要返回的是一个可变字符串类型的值。

另外，与存储过程不同的是，调用函数时，是作为表达式的一部分被调用的。而存储过程调用时，其本身就是一个PL/SQL语句。有关函数的详细讲解可以参看第23章。

### 18.1.5 包

包是由两个相互独立的部分组成的，包头和包体。其中，包头是用来包含包内容的相关信息。只包含有过程和函数的定义，不包含任何过程和函数实现的代码。可以使用关键字CREATE [OR REPLACE] PACKAGE定义一个包头。其中，关键字OR REPLACE是可选的。

```
CREATE OR REPLACE PACKAGE selectStudentPackage  
AS  
    PROCEDURE insert_student(p_stuID VARCHAR2(15), p_stuName VARCHAR2(10), p_age NUMBER(2),
```

```
p_sex VARCHAR2 (20,p_birth DATE);  
FUNCTION select_student(p_stuID VARCHAR2(15)) RETURN VARCHAR2;;
```

包体是用来实现包头中定义的过程和函数。可以使用关键字CREATE [OR REPLACE] PACKAGE BODY定义一个包体。其中，关键字OR REPLACE是可选的。

```
CREATE OR REPLACE PACKAGE BODY selectStudentPackage  
AS  
    PROCEDURE insert_student(  
        p_stuID VARCHAR2(15),  
        p_stuName VARCHAR2(10),  
        p_age NUMBER(2),  
        p_sex VARCHAR2 (20),  
        p_birth DATE)  
    AS  
    BEGIN  
        /*  
        *插入学生信息  
        */  
        INSERT INTO t_student  
            VALUES(p_stuID, p_stuName, p_age, p_sex, p_birth);  
    END;  
    FUNCTION select_student(  
        p_stuID VARCHAR2(15))  
    RETURN VARCHAR2;  
    AS  
        /*  
        *查询学生信息  
        */  
        SELECT stuName  
            INTO v_stuName  
        FROM t_student  
        WHERE stuID = v_stuID;  
        RETURN v_stuName;  
    END;  
END;
```

从这段代码中可以看到，在selectStudentPackage包中，包头部分只是用来定义过程和函数，这里定义了一个insert\_student的存储过程和一个select\_student的函数，select\_student的函数返回值是VARCHAR2类型的。包体部分是对包头中定义的存储过程和函数的实现。在包体中实现的存储过程和函数要和包头中定义的存储过程和函数对应，包括存储过程和函数的名称、参数名称、参数个数以及参数模式等。有关包的详细讲解可以参看第24章。

### 18.1.6 触发器

触发器可以认为是一种特殊类型的存储过程，它是由数据库管理系统隐式执行的。在触发器中可以包含SQL语句和PL/SQL语句，在定义触发器时，还需要指定触发事件，常用的触发事件主要是数据操纵语言DML（例如，INSERT、UPDATE和DELETE）。可以使用关键字CREATE [OR REPLACE] TRIGGER定义一个触发器。其中，关键字OR REPLACE是可选的。



```
CREATE OR REPLACE TRIGGER student_result_update_cascade
AFTER UPDATE OR DELETE OF stuID ON t_student
FOR EACH ROW
BEGIN
    UPDATE t_result
    SET stuID = :new. stuID
    WHERE stuID =:old. stuID;
END;
```

这段PL/SQL语句块使用CREATE OR REPLACE TRIGGER关键字定义一个触发器，用于实现学生信息表和学生成绩表的级联更新操作。当在学生信息表t\_student中执行修改和删除操作的时候，都会触发该触发器。通过定义触发器，实现主从表的级联更新操作，从而保证了数据库的参照完整性。有关触发器的详细讲解可以参看第25章。

## 18.2 PL/SQL基本要素

PL/SQL中所有的程序代码都是由特定的词汇单元组成的。使用PL/SQL编写代码就需要了解这些词汇单元。PL/SQL中的词汇单元可以分为标识符、分隔符、文字和注释。本节就来介绍这四种词汇单元的使用方法和作用。

### 18.2.1 标识符

标识符可以由字母、数字、下划线、货币符号和“#”字符组成，并且必须以字母开头，其最大长度为30个字符。使用标识符可以用来命名PL/SQL语句块中的变量、常量、数据类型、游标、参数、子程序（包括存储过程、函数和包）的名称。例如，下面的标识符都是合法的。

```
A
v123
v_stuID
v$salary
v_error_#
```

如果不是以字母开头，或者使用了除字母、数字、下划线、货币符号和“#”字符以外的符号，或者超出了30个字符的长度，则这样的标识符就是非法的。例如，下面的标识符都是非法的。

1_stuID	-- 以数字开头
字母a	-- 以汉字开头
v/b	-- 含有非法字符 (/)
student Name	-- 含有空格
this_word_is_too_long_and_it_is_more_than30	-- 字符长度超过30

除了上面讲到的标识符定义之外，在PL/SQL语句块中定义变量、常量、数据类型、游标、参数、子程序的名称时，还需要注意以下一些规则。

- PL/SQL语句块中的标识符不区分大小写。例如，变量v\_stuName和变量v\_STUNAME表示的意義相同。
- 程序中的标识符不能使用PL/SQL中的关键字命名。例如，使用begin定义一个变量将会产生一个编译错误。

- ❑ PL/SQL 语句块中使用标识符定义变量或者常量时，一行只能定义一个变量或者常量。在程序中一行定义两个变量或者常量是不允许的。
- ❑ 如果希望标识符中包括除字母、数字、下划线、货币符号和“#”字符以外的符号，或者希望使用关键字作为标识符，或者希望区分标识符的大小写，可以使用双引号（“”）将该标识符引起来。例如，“student Name”、“v/b”、“V/B”、“begin”、“BEGIN”等，这些都是合法的标识符。

**注意** 虽然可以使用双引号（“”）的形式将 PL/SQL 中的关键字引起来，并作为一个合法的标识符来使用，但是在实际应用中，并不提倡使用 PL/SQL 中的关键字作为标识符，而且这并不是一个好的编程规范。

18.2.2 分隔符

分隔符是在 PL/SQL 中含有特殊意义的符号，使用分隔符可以将标识符分隔开来。分隔符可以是单个符号（例如，+、-、\*、/等），也可以是由多个符号组成的（例如，:=、<<、>>等）。PL/SQL 中的分隔符及其表示的含义如表 18.1 所示。

表 18.1 分隔符及其表示的含义

分隔符	含义	分隔符	含义
+	加法操作符	:	语句终止符
-	减法操作符	@	数据库访问操作符
*	乘法操作符	!=或者<>或者^=	不等于操作符
/	除法操作符	>=	大于等于操作符
=	等号操作符	<=	小于等于操作符
>	大于操作符	=>	联结操作符
<	小于操作符	:=	赋值操作符
%	属性提示符		字符串连接符
(	表达式起始分隔符	<<	标记起始分隔符
)	表达式结束分隔符	>>	标记结束分隔符
,	字符串分隔符	--	单行注释符
"	字符标记分隔符	/*	多行注释起始符
.	组件分隔符	*/	多行注释结束符
:	绑定变量提示符	**	幂指数操作符

18.2.3 文字

文字是指数字、字符、日期时间值和布尔值，但是它们并不是标识符。例如，12345、TRUE、FALSE、NULL 等。下面分别对它们进行介绍。

1. 数字文字

数字文本包括整数和浮点数，它们都可以作为算术表达式的一部分来使用，数字文字可以被赋值给 NUMBER 类型的变量。例如，下面的数字文字都是合法的。

## 零基础学SQL

```
123          -- 整数
1.23         -- 浮点数
```

在PL/SQL中，还可以使用幂指数操作符和科学计数法表示数字文字。其中，幂指数操作符为\*\*；科学计数法是在一个浮点数的后面使用字母E（或者e）再加上一个整数的形式来表示。例如，下面的数字文字都是合法的。

```
1.23E5       -- 相当于1.23*100000
1.23E-5      -- 相当于1.23*0.000001
-1.23E5      -- 相当于-1.23*100000
3*10**5      -- 相当于3*100000
```

### 2. 字符文字

字符文字可以由单引号引住的单个字符，这里的单个字符可以是字母、数字或者是其他符号。例如，'A'、'6'、'%'等；也可以是由单引号引住的多个字符，例如，'abc'、'123'等。字符文字可以被赋值为CHAR或者是VARCHAR2类型的变量。

在由单引号引住多个字符时，如果该字符串中包含有单引号（'），那么就需要在该字符串中使用两个单引号来表示。例如，要想输出字符串It's a string，就需要在字符串It's a string 的“s”中并列放置两个单引号。

```
'It''s a string'
```

如果想要得到一个只包含有单引号的字符串，就可以使用下面的方法。

```
''''
```

这里一共使用了4个单引号来表示一个包含有单引号的字符串。其中，第一个单引号指定字符串的开始，最后一个单引号指定字符串的结束，而在第一个单引号和第四个单引号之间的部分表示的就是一个字符串，这个字符串是一个包含有单引号的字符串。

### 3. 日期时间文字

日期文字是指日期时间值。在PL/SQL中，日期时间值也需要使用单引号将其引住，而且日期时间值还需要与数据库中的日期语言和日期格式相匹配。例如，下面的日期时间值都是合法的。

```
'2009-08-31'
'31-8月-09'
```

### 4. 布尔文字

布尔文字包括TRUE、FALSE和NULL三种。布尔文字主要用在IF的条件表达式或者是LOOP的循环语句中，用来判断条件的真或者假。布尔文字可以被赋值给BOOLEAN类型的变量。

#### 18.2.4 注释

PL/SQL中注释包括单行注释和多行注释。注释在编译执行PL/SQL时会被忽略。使用注释可以提高代码的可阅读性，有利于对代码的理解。

##### 1. 单行注释

单行注释主要用于对一行代码进行解释说明，在PL/SQL中，使用“--”来对代码进行代码注释。



例如下面的语句块中，使用的就是单行注释。

```
DECLARE
    v_stuID VARCHAR2(15) := 's145203';      -- 学生编号
    v_stuName VARCHAR2(10) := '杜玉';        -- 学生姓名
    v_age NUMBER(2) := 22;                   -- 学生年龄
    v_sex VARCHAR2(20) := '女';              -- 学生性别
    v_birth DATE := '19870513';              -- 学生出生日期
BEGIN
    -- 插入学生信息
    INSERT INTO t_student
        VALUES(v_stuID, v_stuName, v_age, v_sex, v_birth);
END;
```

## 2. 多行注释

多行注释主要用于对一段代码进行解释说明，在PL/SQL中，使用“/\*\*/”来对代码进行代码注释。例如下面的语句块中，使用的就是多行注释。

```
DECLARE
    v_stuID VARCHAR2(15) := 's145203';      /*学生编号*/
    v_stuName VARCHAR2(10) := '杜玉';        /*学生姓名*/
    v_age NUMBER(2) := 22;                   /*学生年龄*/
    v_sex VARCHAR2(20) := '女';              /*学生性别*/
    v_birth DATE := '19870513';              /*学生出生日期*/
BEGIN
    /*
    *插入学生信息
    *并根据学生编号查询学生姓名
    */
    INSERT INTO t_student
        VALUES(v_stuID, v_stuName, v_age, v_sex, v_birth);
    SELECT stuName
        INTO v_stuName
    FROM t_student
    WHERE stuID = v_stuID;
    DBMS_OUTPUT.PUT_LINE ('学生姓名为' || v_stuName);
END;
```

## 18.3 声明和初始化变量

在PL/SQL中，变量是用来存储数据的，通过变量可以实现与数据库之间的通信。存储在变量中的数据可以插入到数据库的数据表中；数据库中的表列中也可以赋值给指定的变量。每一个变量在定义时都需要指定一个特定的数据类型。声明一个变量的语法规则如下：

```
variable_name datatype [CONSTANT] [NOT NULL] [ :=value[DEFAULT]]
```

其中，variable\_name表示要声明变量的名字；datatype表示该变量的数据类型；CONSTANT表示定义的是常量；NOT NULL表示该变量不能为空，即必须要对其进行初始化；:=表示赋值操作符；value

## 零基础学SQL

表示为该变量赋的初值；DEFAULT表示为变量设定初始值。CONSTANT、NOT NULL、DEFAULT和:=value都是可选的。

在对变量进行初始化操作时，在赋值操作符左边的值必须指向实际的存储单元（例如，变量），在赋值操作符左边的值可以是一个存储单元、字符串内容或者是一个数值。例如，以下的变量声明都是合法的。

```
DECLARE
  v_stuName VARCHAR2(10);
  v_stuID VARCHAR2(15) := 's145203';
  c_insurance_percent CONSTANT NUMBER(3,2) = 14.5;
  v_sum NUMBER(4) NOT NULL:=1234;
  v_avg NUMBER DEFAULT 0;
```

其中，变量v\_stuName的数据类型为VARCHAR2，并指定了其长度为10；变量v\_stuID的数据类型也是VARCHAR2，并为其赋了初值s145203；c\_insurance\_percent由于使用了CONSTANT关键字，所以它表示的是一个常量，在变量声明时要为其赋初值，这里为其赋值为14.5；变量v\_sum使用了NOT NULL关键字，表示该变量不能为空，在变量声明时要为其赋初值，这里为其赋值为1234；变量v\_avg使用了DEFAULT关键字，表示要为该变量设置一个默认值，这里设定的默认值为0。

**注意** 如果在一个变量声明时没有初始化，那么它的默认值将是NULL，可以在BEGIN部分对其赋初值；如果变量声明中指定了NOT NULL，则该变量必须要初始化。对于上面的例子中，如果写成v\_sum NUMBER(4) NOT NULL;则这个声明就是非法的。

在上面的这些合法的声明中可以看到，变量的声明都是在DECLARE部分完成的。在PL/SQL中声明变量时，需要注意下面两个问题。

❑ 在PL/SQL中，每一行只能声明一个变量。例如，下面的变量声明是错误的。

```
DECLARE
  v_stuName, v_stuID VARCHAR2(15);           -- 错误的变量声明，同一行上声明了两个变量
```

❑ 在PL/SQL中，每一个变量只能有一个赋值。例如，下面的变量赋值是错误的。

```
DECLARE
  v_sum1 NUMBER(4);
  v_sum2 NUMBER(4);
BEGIN
  v_sum1:=v_sum2:=1234;                       --错误的变量声明，每一个变量只能有一个赋值
END;
```

## 18.4 PL/SQL数据类型

在前面的PL/SQL的一些例子中，可以看到，在PL/SQL语句块中，当在定义某一个变量、常量、参数的时候，都要为其指定数据类型。例如，v\_stuID VARCHAR2(15);这个语句中定义了一个v\_stuID的变量，并为其指定了变量类型为VARCHAR2类型的。在PL/SQL语句块中，变量类型主要有四种类型：标量类型、复合类型、引用类型和LOB类型。这一节就来介绍这四种数据类型。

### 18.4.1 标量类型

在讲解标量类型之前，先来了解一个标量变量的概念。所谓标量变量，就是指只能存储单个数值的变量。当定义一个标量变量时，就需要为其指定标量数据类型。标量数据类型主要包括数字类型、字符类型、布尔类型、日期时间类型等，每种标量类型又对应包含有相应的子类型。子类型可以认为是类型的别名，一般可以从两个方面考虑使用子类型，一个是考虑与其他数据库的数据类型相兼容，另外一个是从代码的可读性考虑。下面就来介绍这几种标量数据类型。

#### 1. 数字类型

数字类型是用来存储整数或者是浮点数的，它包括以下几种基本的类型，分别是NUMBER、BINARY\_INTEGER、PLS\_INTEGER、BINARY\_FLOAT和BINARY\_DOUBLE类型。其中，NUMBER类型是用来定义整数或者是浮点数的，BINARY\_INTEGER和PLS\_INTEGER类型是用来定义整数的。

定义一个NUMBER类型语法规则如下：

```
NUMBER(P, S)
```

其中，参数P表示精度，用来指定数值中所有数字的总位数；参数S表示刻度，用来指定小数点右边的数字位数。参数S的值可以为负数，如果参数S的值为负数，则表示从小数点开始向左进行舍入计算。参数P和参数S都是可选的，但是如果指定了参数S的值，则必须还要指定参数P的值。

**注意** NUMBER类型数值的最大长度为38位，刻度范围是-84~127。

下面来看几个关于NUMBER类型的例子。

```
v_sum1 NUMBER:=123          -- 变量v_sum1实际存储值为123
v_sum2 NUMBER(4):=1234      -- 变量v_sum2实际存储值为1234
v_sum3 NUMBER(6,2):=1234.56 -- 变量v_sum3实际存储值为1234.56
v_sum4 NUMBER(4,-1):=1234   -- 变量v_sum4实际存储值为1230
v_sum4 NUMBER(4,2):=123.45  -- 错误，参数P的值超出了定义值的精度范围
```

与NUMBER类型等价的子类型主要包括DEC、DECIMAL、DOUBLE PRECISION、FLOAT、NUMERIC、REAL、INT、INTEGER和SMALLINT。其中，DEC、DECIMAL和NUMERIC可以定义最大精度为38位的十进制浮点数；DOUBLE PRECISION和FLOAT可以定义最大精度为126位二进制的浮点数（近似相当于38位十进制数）；REAL可以定义最大精度为63位二进制的浮点数（近似相当于18位十进制数）；INT、INTEGER和SMALLINT可以定义最大精度为38位的十进制整数。

BINARY\_INTEGER是用来定义整数的，它可以用来存储带符号的整数值，其取值范围为-2147483647~2147483647。使用这种数据类型定义的整数可以直接参与算术运算。BINARY\_INTEGER类型也有一些与之等价的子类型，主要包括NATURAL、NATURALN、POSITIVE、POSITIVEN、SIGNTYPE。

与NUMBER类型中子类型的数值不受限制不同，BINARY\_INTEGER的子类型定义的整数的取值范围是有一定限制的。其中，NATURAL类型的取值范围为0~2147483647；NATURALN类型的取值范围与NATURAL类型的取值范围相同，但是，NATURALN类型不允许为整数赋NULL值；POSITIVE类型的取值范围为1~2147483647；POSITIVEN类型的取值范围与POSITIVE类型的取值范围相同，但是，POSITIVEN类型不允许为整数赋NULL值；SIGNTYPE类型的整数只能取3个值，分别是0、1和-1。

PLS\_INTEGER类型的取值范围与BINARY\_INTEGER类型的取值范围相同，都是用来存储带符号

的整数。与BINARY\_INTEGER类型不同的是，在使用PLS\_INTEGER类型的数值进行计算时，如果计算出现溢出，则会抛出异常，而BINARY\_INTEGER类型的数值在计算时，如果计算出现溢出现象，则不会抛出异常。

相对于NUMBER类型和BINARY\_INTEGER类型，PLS\_INTEGER的性能要更好一些。它的计算速度要比NUMBER类型和BINARY\_INTEGER类型更快一些，而且存储空间比NUMBER类型更小。

BINARY\_FLOAT和BINARY\_DOUBLE这两种类型是Oracle新增的数据类型，BINARY\_FLOAT用来定义单精度浮点数，为该类型的变量赋值时，需要带有后缀f（例如，1.2f）；BINARY\_DOUBLE用来定义双精度浮点数，为该类型的变量赋值时，需要带有后缀d（例如，1.23456d）。

## 2. 字符类型

字符类型是用来存储字符数据或者字符串的。字符类型主要包括CHAR、VARCHAR2、NCHAR、NVARCHAR2、LONG、LONG RAW和RAW。

CHAR类型可以用来定义字符串，它可以存储固定长度的字符数据。定义一个CHAR类型的语法规则如下：

```
CHAR[(max_length [CHAR | BYTE] )]
```

其中，参数max\_length用来指定字符串的最大长度，它可以以CHAR（字符）为单位，也可以以BYTE（字节）为单位。其长度范围为1~32767字节（在数据库中，其最大存储长度为2000字节）。参数max\_length是可选的。如果不指定最大长度，其长度的默认值为1。

**注意** 由于CHAR类型定义的是固定长度的字符串，所以如果定义的字符长度比实际指定的长度少，则系统会使用空白字符将其填充。这样在实际使用中，可能会出现使用PL/SQL或者SQL语句对字符进行比较时，出现CHAR类型变量不能匹配的问题。

由于CHAR类型在数据库中，其最大存储长度为2000字节，所以在插入操作中，不能将超过2000字节的数据插入到定义为CHAR类型的数据列中。如果要插入的数据超过2000字节，可以使用VARCHAR2或者LONG定义数据列。CHAR类型也有一个子类型CHARACTER，它与CHAR类型的取值范围是相同的，CHARACTER类型可以兼容NSI/ISO和IBM类型。

VARCHAR2类型可以用来定义可变长度字符串，它可以存储可变长度的字符数据。定义一个VARCHAR2类型的语法规则如下：

```
VARCHAR2 [(max_length [CHAR | BYTE] )]
```

其中，参数max\_length用来指定字符串的最大长度，它可以以CHAR（字符）为单位，也可以以BYTE（字节）为单位。其长度范围为1~32767字节（在数据库中，其最大存储长度为4000字节）。

由于VARCHAR2类型在数据库中，其最大存储长度为4000字节，所以在插入操作中，不能将超过4000字节的数据插入到定义为VARCHAR2类型的数据列中。如果要插入的数据超过4000字节，可以使用LONG或者CLOB定义数据列。VARCHAR2类型有两个子类型STRING和VARCHAR，这两个类型都可以兼容NSI/ISO和IBM类型。

**注意** VARCHAR2和VARCHAR的意义是相同的，但是VARCHAR2类型是由Oracle定义的，而VARCHAR类型是由ANSI定义的。如果在PL/SQL语句块中要定义一个可变长的字符串类型，建议使用VARCHAR2类型。

NCHAR类型是用来存储PL/SQL里国家字符集中的字符数据的。它可以支持任何Unicode字符数据。定义一个NCHAR类型的语法规则如下：

```
NCHAR [(max_length )]
```

其中，参数max\_length 用来指定字符串的最大长度，该长度是以字符为单位来指定的。在AL16UTF16的编码方式下，其最大长度可以为32767/2；在UTF8的编码方式下，其最大长度可以为32767/3。其最大长度是可选的，如果没有选定最大长度，则它的默认长度为1。在数据库中，其最大存储长度为2000字节，在向数据库做插入操作时，向该数据类型的列中插入的内容不能超过2000字节。

**注意** NCHAR类型的参数中，参数max\_length指定的长度只能是以字符为单位。对于CHAR和NCHAR，从CHAR类型可以安全地转换为NCHAR类型，但是从NCHAR类型转换到CHAR类型不一定是安全的，可能会导致数据丢失。

NVARCHAR2也是用来存储PL/SQL里国家字符集中的字符数据的，不过它可以存储可变长的Unicode字符数据。定义一个NVARCHAR2类型的语法规则如下：

```
NVARCHAR2 (max_length )
```

其中，参数max\_length 用来指定字符串的最大长度，该长度是以字符为单位来指定的。在AL16UTF16的编码方式下，其最大长度可以为32767/2；在UTF8的编码方式下，其最大长度可以为32767/3。在数据库中，其最大存储长度为4000字节，在向数据库做插入操作时，向该数据类型的列中插入的内容不能超过4000字节。

**注意** NVARCHAR2类型的参数中，参数max\_length指定的长度只能是以字符为单位。对于VARCHAR2和NVARCHAR2，从VARCHAR2类型可以安全地转换为NVARCHAR2类型，但是从NVARCHAR2类型转换到VARCHAR2类型不一定是安全的，可能会导致数据丢失。

LONG类型也可以用来定义可变长度字符串，它也可以存储可变长度的字符数据。其最大长度为32760字节。在数据库中，其最大存储长度为2GB。定义为LONG类型的列可以进行SELECT、INSERT、UPDATE等操作，但是不能应用在表达式、SQL子句（例如，WHERE、GROUP BY等）或者SQL函数中，否则，系统就会报错。

**注意** 在SQL语句中，PL/SQL会将LONG类型的值转换为VARCHAR2类型。如果被绑定的VARCHAR2值超过4000字节，Oracle会自动地将其转换LONG类型。虽然，Oracle可以将超过4000字节的值自动转换为LONG类型，但是由于LONG类型不能应用在SQL函数中，所以此时系统会报错。

如果先要存储二进制数据或二进制字符串，可以使用LONG RAW类型。其最大长度也是32760字节。由于在数据库中，可以使用LONG RAW类型的最大长度，所以可以把一个在PL/SQL语句块中定义的LONG RAW类型的变量插入到数据库的指定列中。在Oracle9i及其以后的版本中，LONG和LONG RAW类型的变量可以与LOB类型的变量交互使用。

**说明** 在Oracle9i及其以后的版本中，建议将LONG类型和LONG RAW类型的变量转换成对应的COLB和BLOB类型的变量。



## 零基础学SQL

RAW类型是用来定义二进制数据或者字节串的，它可以用来存储二进制数据或者字节串。其语法规则如下：

```
RAW(max_length)
```

其中，参数max\_length 用来指定字节串的最大长度，其长度范围为1~32767字节。在数据库中RAW类型字段的最大长度是2000个字节。

### 3. 布尔类型

布尔类型是用来定义布尔变量的。变量值包括TRUE、FALSE和NULL。布尔变量主要是用来进行逻辑操作的，在布尔变量中只允许使用逻辑操作符。

**注意** 在数据库的数据列中，不能使用布尔类型，该类型只用在PL/SQL语句块中。

### 4. 日期时间类型

日期时间类型主要是用来定义日期时间数据的，主要包括DATE、TIMESTAMP和INTERVAL三种类型。

DATE类型是用来定义日期时间数据的，它可以存储定长的日期时间。其日期的有效范围为公元前4721年1月1日到公元9999年12月31日。其默认值在日期部分为当月的第一天，时间部分为午夜时间。可以使用SYSDATE函数取得系统当前日期（可以参看10.3.1节）。PL/SQL语句块中，会将默认日期格式的字符值转换为DATE类型。在Oracle数据库中，可以使用NLS\_DATE\_FORMAT设置日期的默认格式。日期还可以进行加减运算（可以参看10.3.2节）。

TIMESTAMP类型也可以存储日期时间数据，它除了可以存储年、月、日、时、分、秒之外，还可以存储秒的小数部分。其语法规则如下：

```
TIMESTAMP[(precision)]
```

其中，参数precision用来指定秒的小数部分的精度，其取值范围为0~6，默认值为6。该参数是可选的。

除了可以指定日期时间秒的小数部分之外，还可以存储日期时间的时区。指定日期时间的时区的TIMESTAMP类型的语法格式如下：

```
TIMESTAMP[(precision)] WITH TIME ZONE
```

其中，参数precision用来指定秒的小数部分的精度，其取值范围为0~6，默认值为6，它是可选的。关键字WITH TIME ZONE用来表示可以指定日期时间的时区位移。

```
DECLARE
  v_time TIMESTAMP ( 3 ) WITH TIME ZONE;
BEGIN
  v_time:= '2009-09-01 09:30:43.104 +03:00';
  -- 执行代码
END;
```

这段PL/SQL的代码中，定义了一个日期时间类型的变量v\_time，并使用WITH TIME ZONE关键字，表示可以指定日期时间的时区位移。这里指定的时区位移为3.00。除了可以使用数字来指定时区位移之外，还可以使用符号名称来指定。例如，"US/Pacific"等。

还可以使用TIMESTAMP WITH LOCAL TIME ZONE类型存储当前数据库的时区。其语法格式如下：

```
TIMESTAMP[(precision)] WITH LOCAL TIME ZONE
```

其中，参数precision用来指定秒的小数部分的精度，其取值范围为0~6，默认值为6，它是可选的。关键字WITH LOCAL TIME ZONE用来表示可以指定数据库的时区。当向数据库插入TIMESTAMP WITH LOCAL TIME ZONE类型数据时，其数据就会被转换为当前数据库的时区。

INTERVAL类型是用来存储两个时间之间的时间间隔的。可以使用INTERVAL YEAR TO MONTH类型存储年月的时间间隔。其语法格式如下：

```
INTERVAL YEAR[(precision)] TO MONTH
```

其中，参数precision表示指定间隔的年数，其取值范围为1~4，默认值为2，该参数是可选的。

另外，还可以使用INTERVAL DAY TO SECOND数据类型存储日和秒的时间间隔。其语法格式如下：

```
INTERVAL DAY[(leading_precision)] TO SECOND[(seconds_precision)]
```

其中，参数leading\_precision用来指定天数；参数seconds\_precision用来指定秒数，其取值范围在0~9。

日期时间类型也有自己的子类型。其子类型主要包括TIMESTAMP\_UNCONSTRAINED、TIMESTAMP\_TZ\_UNCONSTRAINED、TIMESTAMP\_LTZ\_UNCONSTRAINED、YMINTERVAL\_UNCONSTRAINED和DSINTERVAL\_UNCONSTRAINED，它们都使用的是最大精度。由于某些日期时间类型的默认精度会比其对应的最大精度小，在进行赋值或者参数传递时，日期时间值可能会丢失精度，为了避免这类问题，可以使用其子类型。

有关标量变量的定义和使用可以参看18.5节。

### 18.4.2 复合类型

所谓复合变量，就是指可以存储多个数值的变量。当定义一个复合变量时，就需要为其指定复合数据类型。复合类型的变量中可以包含一个标量变量，也可以包含多个标量变量。复合类型主要包括记录、表、嵌套表和变长数组四种类型。

在PL/SQL的记录中可以包含多个变量。在使用记录类型时，需要在DECLARE部分定义记录类型和记录变量。它类似于C语言中的结构。有关PL/SQL记录的定义和使用可以参看18.6.1小节。

在PL/SQL中，表类似于C语言中的数组。但是PL/SQL表中下标的使用没有上下限的要求，下标可以为负数，而且表中的元素个数没有限制。它不能作为数据库的表中数据列的数据类型。有关PL/SQL表的定义和使用可以参看18.6.2节。

在PL/SQL中，嵌套表也与C语言中的数组相类似，但是PL/SQL嵌套表中，其元素下标是以1开始的，对元素个数是没有限制的，因此嵌套表的大小是可以自动增长的。与C语言中的数组不同，在嵌套表中，其下标索引可以是不连续的。另外，它还可以作为数据库的表中数据列的数据类型。有关PL/SQL嵌套表的定义和使用可以参看18.6.3节。

在PL/SQL中，变长数组可以存储固定数量的元素，在定义时需要为其指定变长数组中元素的最大个数，在运行时可以改变其大小，但是其改变的大小不能超过定义变长数组时指定的上限（即在定义变长数组时指定的最大个数）。与嵌套表相类似，它也可以作为数据库的表中数据列的数据类型。有关变长数组变量的定义和使用可以参看18.6.4节。

### 18.4.3 引用类型

在PL/SQL中，当定义标量变量或者是复合变量时，内存会为其分配适当的存储空间供应用程序来引用，在该变量的作用域结束之前，为其分配的存储空间不会被释放。为了降低内存空间的占有率，实现相同对象的共享，就需要使用引用类型。

在PL/SQL中的引用类型类似于C语言中的指针。引用类型主要包括REF CURSOR和REF两种类型。其中，REF CURSOR是游标变量，REF表示对象类型，它可以指向一个对象，是一个指向对象实例的指针。有关游标的操作可以参看第20章。

### 18.4.4 LOB类型

LOB数据类型是可以用来存储大型对象的变量的。大型对象可以是二进制数据（例如，图形、图形、视频、音频等），也可以是字符数据，其最大存储长度可以为4GB。除此之外，它还可以高效地对数据进行分段访问，其效率要比LONG、LONG ROW等数据类型更高。如果想对LOB数据类型进行处理，可以通过使用Oracle中的系统包DBMS\_LOB。

LOB数据类型可以分为BFILE、BLOB、CLOB和NCLOB几种类型。其中，BFILE数据类型可以存储二进制对象，其存储内容被存储在OS（操作系统）的文件中，而不是存储在数据库中。另外，BFILE类型并不支持事务操作，它是只读的，其大小最大为4GB。BLOB、CLOB和NCLOB数据类型也可以存储二进制对象，其存储的内容在数据库中，存储的大小最大为4GB。这三种数据类型支持事务操作。

## 18.5 定义和使用标量变量

标量变量只能存储单个的数值。标量数据类型主要包括数字类型、字符类型、布尔类型、日期时间类型等。本节就来介绍如何在PL/SQL语句块中定义和使用标量变量。

### 18.5.1 定义标量变量

PL/SQL语句块中，在使用标量变量之前，需要先在DECLARE（定义部分）中对其进行定义，定义之后才能在BEGIN（执行部分）或者EXCEPTION（异常处理部分）中使用它们。定义标量变量的语法规则如下：

```
variable_name [CONSTANT] datatype [NOT NULL] [:=value[DEFAULT]]
```

其中，variable\_name表示指定的变量名；CONSTANT表示定义的是常量；datatype用于指定定义的变量（或者常量）的数据类型；NOT NULL表示不为空的约束条件；DEFAULT表示为变量设定初始值；:=value用来指定变量（或者常量）的初始值。CONSTANT、NOT NULL、DEFAULT和:=value这些都是可选的。

```
v_age NUMBER(2);  
v_teaID VARCHAR2(15) NOT NULL;  
v_stuDate DATE;  
c_insurance_percent CONSTANT NUMBER (3,2) = 14.5;  
v_valid BOOLEAN NOT NULL DEFAULT TRUE;
```

以上这些都是合法的标量变量的定义。其中，c\_insurance\_percent表示定义的是一个常量，v\_valid

表示定义的是一个布尔类型的变量，该变量的默认值为TRUE。

### 18.5.2 使用标量变量

在DECLARE（定义部分）中对标量变量定义之后，就可以在BEGIN（执行部分）或者EXCEPTION（异常处理部分）使用这些标量变量了。下面来看一个使用标量变量的例子。

例18.1 使用标量变量查询指定学生信息。

```
DECLARE
    v_stuName VARCHAR2(10) ;           -- 学生姓名
    v_age NUMBER(2);                   -- 学生年龄
    v_sex VARCHAR2 (20);               -- 学生性别
    v_birth DATE;                      -- 出生日期
BEGIN
    /*
    *根据输入的学生编号查询学生信息
    */
    SELECT stuName, age, sex, birth
        INTO v_stuName, v_age, v_sex, v_birth
    FROM t_student
    WHERE stuID = &sid;                -- 将输入的学生编号作为限定条件
    DBMS_OUTPUT.PUT_LINE ('学生姓名为' || v_stuName);      -- 显示学生姓名
    DBMS_OUTPUT.PUT_LINE ('学生年龄为' || v_age);          -- 显示学生年龄
    DBMS_OUTPUT.PUT_LINE ('学生性别为' || v_sex);          -- 显示学生性别
    DBMS_OUTPUT.PUT_LINE ('出生日期为' || v_birth);        -- 显示出生日期
END;
```

这段PL/SQL语句是要求将用户在屏幕上输入的学生编号作为查询的限定条件，查询指定学生编号的学生信息。其中，SELECT语句中使用了INTO关键字，表示这个SELECT语句返回的结果只有一条数据。&sid表示要求用户在屏幕上输入一个学生编号。其查询结果如下：

```
输入sid的值: s102203
学生姓名为: 赵亮
学生年龄为: 23
学生性别为: 男
出生日期为: 1986-05-16
```

#### 注意

在使用标量变量取得SELECT语句中的查询结果时，其标量变量的个数要与SELECT语句中指定的数据列的个数相同，并且标量变量中的数据类型和长度要与数据表中相应列的数据类型和长度相匹配。在使用SELECT INTO语句对数据表进行查询时，其返回的结果应该只能有一条数据，如果返回的结果多于一条数据，则系统会报错。

### 18.5.3 使用%TYPE属性绑定变量

在使用标量变量取得SELECT语句中的查询结果时，其标量变量中的数据类型和长度要与数据表中相应列的数据类型和长度相匹配。如果在例18.1中，如果将数据表中列stuID中的长度变为10，此时DECLARE部分定义的标量变量v\_stuID中定义的数据类型VARCHAR2指定的长度还是15，这样，数据表中列的长度和PL/SQL语句中定义的标量变量v\_stuID指定的长度就不匹配了。当输入的sid值的长度

## 零基础学SQL

超过10时，系统就会出现VALUE\_ERROR未经处理的预定义异常。（有关VALUE\_ERROR预定义异常内容可以参看21.4.2节）

为了避免由于修改数据表中某一个数据列的数据类型或者长度而使PL/SQL程序出现类似上面的运行错误，可以使用%TYPE属性定义标量变量。%TYPE属性会根据数据表中列的数据类型和长度自动匹配新定义的变量。例如，对于例18.1，使用%TYPE属性就可以这样完成。

```
DECLARE
    v_stuName    t_student.stuName%TYPE;    -- 学生姓名
    v_age        t_student.age%TYPE;        -- 学生年龄
    v_sex        t_student.sex%TYPE E;      -- 学生性别
    v_birth      t_student.birth%TYPE;      -- 出生日期
BEGIN
    /*
    *根据输入的学生编号查询学生信息
    */
    SELECT stuName, age, sex
        INTO v_stuName, v_age, v_sex
    FROM t_student
    WHERE stuID = &sid;                    -- 将输入的学生编号作为限定条件
    DBMS_OUTPUT.PUT_LINE ('学生姓名为' || v_stuName);    -- 显示学生姓名
    DBMS_OUTPUT.PUT_LINE ('学生年龄为' || v_age);        -- 显示学生年龄
    DBMS_OUTPUT.PUT_LINE ('学生性别为' || v_sex);        -- 显示学生性别
    DBMS_OUTPUT.PUT_LINE ('出生日期为' || v_birth);      -- 显示出生日期
END;
```

这段PL/SQL语句中在DECLARE部分，原来定义的标量变量的数据类型都使用了%TYPE属性来替代原来指定的数据类型。这样，就不会出现因为修改数据表中某一个或者某些列的数据类型或者长度，而造成数据表中列的数据类型或者长度与PL/SQL语句中定义的对应的标量变量指定的数据类型或者长度不匹配的现象了。

在实际应用中，如果要通过变量保存数据表中的数据，应该使用%TYPE声明变量。通过使用%TYPE属性绑定变量，可以使程序在重新编译之后，自动适应数据表中的变化，使用数据表中新的数据类型。它可以提高PL/SQL程序的执行效率，同时也可以保证PL/SQL程序的健壮性。

在PL/SQL中除了可以使用%TYPE属性绑定变量，还可以使用%ROWTYPE属性对记录与数据表、视图或者游标进行绑定。有关%ROWTYPE属性的用法可以参看18.6.1节。

## 18.6 定义和使用复合类型

复合类型的变量中可以包含一个标量变量，也可以包含多个标量变量。复合数据类型可以用来处理多行多列的数据。复合类型主要包括记录、表、嵌套表和变长数组四种类型。这一节就来介绍如何在PL/SQL语句块中定义和使用复合类型。

### 18.6.1 定义和使用记录

在引用记录变量之前，首先需要在DECLARE部分定义记录类型和记录变量。定义记录类型和记录变量的语法规则如下：



```
TYPE type_name IS RECORD(  
    record_declaration[,  
    record_declaration  
    ...]  
);  
record_name type_name
```

其中，type\_name表示记录类型名称；record\_declaration表示记录变量，记录变量可以有一个，也可以有多个，多个记录变量之间要用逗号分开；record\_name表示记录变量的名称。如果想定义一个记录类型和记录变量，可以使用下面的方法实现。

```
DECLARE  
    TYPE t_studetRec IS RECORD(  
        v_stuName      t_student . stuName%TYPE ,           -- 学生姓名  
        v_age          t_student . age%TYPE,                -- 学生年龄  
        v_sex          t_student . sex%TYPE E,               -- 学生性别  
        v_birth        t_student . birth%TYPE,               -- 出生日期  
    );  
    student_record t_studetRec;  
BEGIN  
    -- 执行代码  
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了记录类型和记录变量。其中，t\_studetRec表示记录类型，在这个记录中，一共包括4个表示学生信息的记录变量，分别是v\_stuName、v\_age、v\_sex和v\_birth。

在DECLARE部分定义了记录类型和记录变量之后，就可以在BEGIN或者EXCEPTION引用记录变量了。在记录中，如果想引用该记录中的某一个记录成员，需要在该记录变量后面加上一个点号，后面再跟上需要引用的记录成员。引用记录变量的语法规则如下：

记录变量.记录成员

例如，对于student\_record这个记录变量，如果想引用该记录变量中的记录成员v\_stuName，就可以使用如下的方法完成。

student\_record.v\_stuName

这就表示引用记录变量student\_record中的表示学生姓名的记录变量v\_stuName。下面通过一个引用记录变量的例子来看一下记录变量是如何使用的。

**例18.2** 使用记录查询指定学生编号的学生信息。

```
DECLARE  
    TYPE t_studetRec IS RECORD(  
        v_stuName      t_student . stuName%TYPE ,           -- 学生姓名  
        v_age          t_student . age%TYPE,                -- 学生年龄  
        v_sex          t_student . sex%TYPE E,               -- 学生性别  
        v_birth        t_student . birth%TYPE,               -- 出生日期  
    );  
    student_record t_studetRec;                               -- 指定记录变量名称  
BEGIN
```

## 零基础学SQL

```
SELECT stuName, age, sex, birth
  INTO student_record
  FROM t_student
 WHERE stuID = &sid;                                -- 根据输入的学生编号作为限定条件
DBMS_OUTPUT.PUT_LINE ('学生姓名为' || student_record.v_stuName); -- 显示学生姓名
DBMS_OUTPUT.PUT_LINE ('学生年龄为' || student_record.v_age);      -- 显示学生年龄
DBMS_OUTPUT.PUT_LINE ('学生性别为' || student_record.v_sex);      -- 显示学生性别
DBMS_OUTPUT.PUT_LINE ('出生日期为' || student_record.v_birth);    -- 显示出生日期
END;
```

这段PL/SQL语句也是要求将用户在屏幕上输入的学生编号作为查询的限定条件，查询指定学生编号的学生信息。在DECLARE部分定义了记录类型和记录变量，在BEGIN部分，根据输入的学生编号作为限定条件查询指定的学生信息，并通过引用记录中的记录变量将学生信息一一显示出来。其查询结果如下：

```
输入sid的值: s115263
学生姓名为: 王海
学生年龄为: 23
学生性别为: 男
出生日期为: 1986-08-02
```

类似于%TYPE，在定义记录类型和记录变量时，可以使用%ROWTYPE属性对记录与数据表、视图或者游标进行绑定。%ROWTYPE属性返回的是一个基于表定义的类型。例如，在记录变量名称为student\_record中使用%ROWTYPE属性，就可以使用下面这行语句来完成。

```
student_record t_student %ROWTYPE;
```

当在student\_record中使用%ROWTYPE属性后，其记录变量的数据类型和名称将与t\_student表中的列的数据类型和名称相对应。如果在实际应用中，t\_student表定义发生了改变，那么使用了%ROWTYPE属性的记录student\_record也会随之改变。

### 18.6.2 定义和使用表

在引用PL/SQL表变量之前，首先需要在DECLARE部分定义PL/SQL表类型和表变量。定义表类型和表变量的语法规则如下：

```
TYPE type_name IS TABLE OF element_type
[NOT NULL] INDEX BY key_type;
table_name type_name;
```

其中，type\_name表示定义的PL/SQL表类型；element\_type表示指定PL/SQL表的数据类型，可以使用%TYPE属性或者%ROWTYPE属性对PL/SQL表类型进行引用；关键字NOT NULL表示不能引用NULL元素，它是可选的；key\_type用来指定PL/SQL表下标的数据类型，PL/SQL表下标的数据类型可以使用VARCHAR2、BINARY\_INTEGER和PLS\_INTEGER；table\_name表示PL/SQL表的变量名称。

**注意** Oracle9i及其以后的版本PL/SQL表下标的数据类型可以使用VARCHAR2；Oracle9i之前的版本PL/SQL表下标的数据类型只能使用BINARY\_INTEGER和PLS\_INTEGER。

如果想定义一个PL/SQL表类型和表变量，可以使用下面的方法实现。

```
DECLARE
    TYPE stu_table_type IS TABLE OF t_student.stuName%TYPE
    INDEX BY BINARY_INTEGER;
    stu_table stu_table_type;
BEGIN
    -- 执行代码
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了PL/SQL表类型和表变量。其中，stu\_table\_type表示PL/SQL表类型的名称，t\_student.stuName%TYPE指定PL/SQL表类型的数据类型；下标的数据类型定义为BINARY\_INTEGER；stu\_table表示PL/SQL表的变量名称。

在DECLARE部分定义PL/SQL表类型和表变量之后，就可以在BEGIN部分和EXCEPTION部分使用PL/SQL表来引用PL/SQL表变量了。例如下面这个例子。

**例18.3** 使用PL/SQL表存储和并输出指定的学生信息。

```
DECLARE
    TYPE stu_table_type IS TABLE OF t_student.stuName%TYPE
    INDEX BY BINARY_INTEGER;
    stu_table stu_table_type ;
BEGIN
    SELECT stuName
    INTO stu_table(1)
    FROM t_student
    WHERE stuID = &sid;
    DBMS_OUTPUT.PUT_LINE ('学生姓名为' || stu_table(1));
END;
```

这段PL/SQL语句块中，在BEGIN部分使用stu\_table(1)存储学生信息，并使用DBMS\_OUTPUT.PUT\_LINE将存储在PL/SQL表变量名称为stu\_table中的学生姓名显示出来，其查询结果如下所示。

输入sid的值：s206363  
学生姓名为：张明

**注意** PL/SQL的表中下标的使用没有上下限的要求，下标可以为负数，例如，对于例18.3，也可以用stu\_table(-1)来存储学生姓名信息，而且表中的元素个数没有限制。它不能作为数据库的表中数据列的数据类型。

### 18.6.3 定义和使用嵌套表

在引用嵌套表变量之前，首先需要在DECLARE部分定义嵌套表类型和变量。定义嵌套表的语法规则如下：

```
TYPE type_name IS TABLE OF element_type
[NOT NULL];
table_name type_name;
```

其中，type\_name表示定义的嵌套表类型；element\_type表示指定嵌套表中元素的数据类型，该数据类型可以使用%TYPE属性，也可以是用户自定义的对象类型，但是有些数据类型不能作为嵌套表中

## 零基础学SQL

元素的数据类型。例如，BINARY\_INTEGER、PLS\_INTEGER、BOOLEAN、NCHAR、NCLOB、NVARCHAR2、REF CURSOR等；关键字NOT NULL表示在嵌套表中的元素不能是NULL，它是可选的；table\_name表示嵌套表的变量名称。

读者可能注意到了，定义嵌套表的语法规则与定义PL/SQL表的语法规则相比，只是比定义PL/SQL表的语法规则少了一个INDEX BY key\_type子句。

**注意** PL/SQL的嵌套表中，其元素下标是以1开始的，对元素个数是没有限制的。

如果想定义一个PL/SQL嵌套表，可以使用下面的方法实现。

```
DECLARE
    TYPE char_table_type IS TABLE OF VARCHAR2(1)
    char_tab char_table_type;
BEGIN
    -- 执行代码
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了PL/SQL嵌套表。char\_table\_type表示PL/SQL嵌套表的类型，VARCHAR2(1)指定PL/SQL嵌套表元素中的数据类型；char\_tab表示PL/SQL嵌套表的变量名称。定义完PL/SQL嵌套表后，在PL/SQL的BEGIN部分使用这个嵌套表，看一下会有什么样的结果产生。其代码如下：

```
DECLARE
    TYPE char_table_type IS TABLE OF VARCHAR2(1)
    char_tab char_table_type;
BEGIN
    char_tab(1):='a';
END; -- 向未经初始化的嵌套表中赋值
```

这段PL/SQL语句块中，在DECLARE部分定义了PL/SQL嵌套表char\_tab，在BEGIN部分向这个嵌套表中添加一个元素字符a，此时系统会产生一个错误提示。

ORA-6531:Reference to uninitialized collection

这个错误提示表示对未经初始化的集合进行了引用。看到这里，读者可能会有这样一个问题，在定义和使用PL/SQL表的时候，当定义完一个PL/SQL表时，向PL/SQL表中添加元素是正常的，为什么使用同样的方法向PL/SQL嵌套表中添加元素，会出错呢？要回答这个问题，需要了解PL/SQL表和PL/SQL嵌套表的创建过程。当定义了一个PL/SQL表后，该表是一个空表，表中开始没有任何元素，所以可以向PL/SQL表中添加元素。而定义了PL/SQL嵌套表后，该表中虽然开始也没有任何元素，但是同PL/SQL其他数据类型一样，PL/SQL嵌套表会被自动地赋上初始值NULL，此时当把一个元素添加到NULL的嵌套表时，系统就会提示上述的错误。（可以参看19.1.2小节，那里使用IF-THEN-ELSE条件分支语句验证了上述说法）

既然不能够在定义完一个PL/SQL嵌套表之后直接为其赋值，那么有什么办法可以初始化一个PL/SQL嵌套表呢？这就需要使用PL/SQL中的构造方法。

### 例18.4 使用PL/SQL嵌套表。

```
DECLARE
    TYPE char_table_type IS TABLE OF VARCHAR2(1)
```

```
char_tab char_table_type := char_table_type('a','b','c') ;-- 初始化嵌套表
BEGIN
    DBMS_OUTPUT.PUT_LINE (char_tab(1)||' ');           -- 显示嵌套表中第一个元素的信息
    DBMS_OUTPUT.PUT_LINE (char_tab(2)||' ');           -- 显示嵌套表中第二个元素的信息
    DBMS_OUTPUT.PUT_LINE (char_tab(3)||' ');           -- 显示嵌套表中第三个元素的信息
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了PL/SQL嵌套表char\_tab，并且对其进行了初始化操作，这里使用了带参数的构造方法初始化了char\_tab。在BEGIN部分，使用Oracle数据库提供的系统包DBMS\_OUTPUT将初始化嵌套表中的字符信息一一输出。其输出结果如下所示。

```
a b c
```

除了使用带参数的构造方法对PL/SQL嵌套表进行初始化之外，也可以使用不带参数的构造方法，如果使用不带参数的构造方法初始化PL/SQL嵌套表，则初始化完成之后，PL/SQL嵌套表就是一个没有任何元素的空表，可以在BEGIN部分向这个空表中添加相应的元素。

在实际使用中，也可以使用EXTEND方法增加PL/SQL嵌套表的长度。例如，如果想在PL/SQL语句块中改变嵌套表char\_tab的长度，就可以使用下面的语句完成。

```
char_tab.EXTEND(10)
```

注意 当在PL/SQL中使用嵌套表时，需要首先使用构造方法对其进行初始化操作，然后才能引用PL/SQL嵌套表中的变量。如果PL/SQL嵌套表未经初始化而直接向其中添加元素，就会得到一个错误提示。

另外，PL/SQL嵌套表还可以作为数据库的表中数据列的数据类型。当使用PL/SQL嵌套表作为数据表中数据列的数据类型时，首先需要使有CREATE TYPE命令创建一个嵌套表类型，而且还要在创建数据表时以嵌套表作为数据类型的列指定一个存储表。在PL/SQL中，创建嵌套表类型并使用嵌套表作为数据表中数据列的数据类型的步骤如下：

(1) 使用CREATE TYPE命令创建一个嵌套表类型。

```
CREATE TYPE stu_address_type IS TABLE OF VARCHAR(50);
```

这里使用CREATE TYPE命令创建了一个嵌套表类型。其中，stu\_address\_type表示PL/SQL嵌套表类型；VARCHAR(50)指定了嵌套表中元素的数据类型。

(2) 使用嵌套表作为数据表中数据列的数据类型，并为该数据表指定存储表。

```
CREATE TABLE studentTable (
    stuID VARCHAR2(15) PRIMARY KEY,
    stuName VARCHAR2(10) NOT NULL,
    stuAddress stu_address_type
) NESTED TABLE address STORE AS address_table
```

其中，在studentTable这张数据表中，数据列stuAddress被定义为嵌套表类型，在该嵌套表中每一个元素存放的是一个学生的地址。在CREATE TABLE创建语句之后，使用NESTED TABLE为stuAddress列指定了一个存储表address\_table。



#### 18.6.4 定义和使用变长数组

变长数组主要是用来处理PL/SQL中的集合。变长数组中元素的个数是受限的，与嵌套表相类似，它也可以作为数据库的表中数据列的数据类型。定义变长数组的语法规则如下：

```
TYPE type_name IS VARRAY (max_size) OF element_data [NOT NULL]
```

其中，type\_name指定变长数组的类型名；max\_size指定变长数组中元素的最大个数，它是一个整数；element\_data指定变长数组的数据类型，该数据类型可以使用%TYPE属性，也可以是用户自定义的对象类型，但是有些数据类型不能作为嵌套表中元素的数据类型。例如，BINARY\_INTEGER、PLS\_INTEGER、BOOLEAN、NCHAR、NCLOB、NVARCHAR2、REF CURSOR等；关键字NOT NULL表示在嵌套表中的元素不能是NULL，它是可选的。

**注意** 在定义一个变长数组时，必须指定其长度。

如果想定义一个变长数组，可以使用下面的方法实现。

```
DECLARE
    TYPE int_array_type IS VARRAY(10) OF NUMBER;
BEGIN
    -- 执行代码
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了变长数组。int\_array\_type表示变长数组的类型，VARRAY(10)指定变长数组的上限，即该变长数组中放置的元素个数最多为10个，NUMBER指定变长数组中元素的数据类型。

同嵌套表一样，在使用变长数组之前，也要对它进行初始化，可以使用构造方法对变长数组的变量进行初始化。

**例18.5** 使用变长数组。

```
DECLARE
    TYPE int_array_type IS VARRAY(10) OF NUMBER;
    IntArray := int_array_type(1,2,3);           -- 初始化变长数组
BEGIN
    DBMS_OUTPUT.PUT_LINE (IntArray (1)|| ' '); -- 显示变长数组中第一个元素的信息
    DBMS_OUTPUT.PUT_LINE (IntArray (2)|| ' '); -- 显示变长数组中第二个元素的信息
    DBMS_OUTPUT.PUT_LINE (IntArray (3)|| ' '); -- 显示变长数组中第三个元素的信息
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了PL/SQL变长数组IntArray，并且对其进行了初始化操作，这里使用了带参数的构造方法初始化了IntArray。在BEGIN部分，使用Oracle数据库提供的系统包DBMS\_OUTPUT将初始化嵌套表中的字符信息一一输出。其输出结果如下所示。

```
1 2 3
```

在实际使用中，可以使用EXTEND方法改变变长数组的长度，但是其改变的大小不能超过定义变长数组时指定的上限。例如，如果想在PL/SQL语句块中改变变长数组IntArray的长度，就可以使用下面的语句完成。

```
IntArray.EXTEND(5);
```

原来变长数组IntArray只能存放3个元素，通过使用EXTEND方法，现在变长数组IntArray中可以存放5个元素。但是在变长数组中使用EXTEND方法，其参数最大不能超过变长数组指定的上限值，这里EXTEND方法中参数的最大值不能超过10。

**注意** 由于在变长数组中指定了数组长度的上限，因此在构造方法对变长数组进行初始化时，其构造方法的参数个数必须小于或者等于变长数组指定的上限。

另外，变长数组还可以作为数据库的表中数据列的数据类型。当使用变长数组作为数据表中数据列的数据类型时，首先也需要使用CREATE TYPE命令创建一个变长数组类型，在PL/SQL中，创建变长数组类型并使用变长数组作为数据表中数据列的数据类型的步骤如下：

(1) 使用CREATE TYPE命令创建一个变长数组类型。

```
CREATE TYPE stu_address_array IS TABLE OF VARCHAR(50);
```

这里使用CREATE TYPE命令创建了一个变长数组类型。其中，stu\_address\_array表示PL/SQL变长数组类型；VARCHAR(50)指定了变长数组中元素的数据类型。

(2) 使用变长数组作为数据表中数据列的数据类型。

```
CREATE TABLE studentTable (  
    stuID VARCHAR2(15) PRIMARY KEY,  
    stuName VARCHAR2(10) NOT NULL,  
    stuAddress stu_address_array  
)
```

其中，在studentTable这张数据表中，数据列stuAddress被定义为变长数组类型，在该变长数组中每一个元素存放的是一个学生的地址。

## 18.7 定义和使用子类型

子类型可以认为是类型的别名，它是对现有基类的扩展。一般定义和使用子类型主要基于两个方面的考虑，一是考虑与其他数据库的数据类型兼容性，另外一个考虑是考虑代码的可读性。这一节就来介绍子类型的定义和使用方法。

### 18.7.1 定义子类型

PL/SQL中的STANDARD包中定义了不少子类型。例如，CHARACTER、INTEGER、DECIMAL等。定义一个子类型的语法规则如下：

```
SUBTYPE subtype_name IS original_type[(constraint)] [NOT NULL];
```

其中，subtype\_name用来指定子类型的名字；original\_type用来指定基类型的名字，它可以是任何的标量类型或者是用户定义的子类型；参数constraint用来表示限定基类的取值范围，它是可选的。

```
DECLARE  
    SUBTYPE t_count IS NUMBER(6);  
    SUBTYPE t_stuDate IS DATE NOT NULL;  
    SUBTYPE t_id IS t_student.stuID %TYPE;  
BEGIN
```

## 零基础学SQL

```
-- 执行代码
END;
```

在这段PL/SQL语句块中，自定义了三个子类型。其中，t\_count基于基类NUMBER，并给定了一个取值范围；t\_stuDate基于基类DATE，并给定了一个NOT NULL的约束条件；t\_id是基于一个“%TYPE”的引用。

**注意** 可以使用%TYPE或者是%ROWTYPE来指定基类型。在用户自定义子类型时，当使用%TYPE表示数据库中表列的数据类型时，其约束条件子类型并不能继承。例如，上例中t\_id是基于一个“%TYPE”的引用，但是它并不能继承stuID列中的NOT NULL约束。

### 18.7.2 使用子类型

在18.7.1节中，介绍了定义子类型的方法。当一个子类型定义完成之后，就可以使用该类型定义其他的变量或者常量了。

```
DECLARE
    SUBTYPE t_sum IS NUMBER;           -- 用户自定义子类型
    v_total t_sum(4);                  -- 定义变量v_total
BEGIN
    v_total := 1000;                  -- 使用变量v_total
    -- 执行代码
END;
```

这段PL/SQL语句块的DECLARE部分定义了一个变量v\_total，其指定的数据类型是一个用户自定义的子类型。其中，t\_sum是一个用户自定义子类型，它是基于基类型NUMBER的；v\_total是定义的一个变量，它的数据类型是t\_sum类型的。在BEGIN部分就可以使用这个变量v\_total了。

**说明** 如果多个不同的子类型都是基于相同的基类型或者是基于的基类型属于同一个数据类型的种类，那么这几个子类型之间是可以交互使用的。

## 18.8 变量的作用域

同其他的高级语言（例如，C、Java等）一样，在PL/SQL中，每一个变量也有自己的作用域。所谓变量的作用域，就是指程序中可以访问到该变量的部分。变量定义在不同的位置上，会有不同的作用域。在PL/SQL中，一个变量的作用域是从变量声明（DECLARE）到该语句块结束（END）的部分。来看下面这个PL/SQL代码段。

```
(1) DECLARE
(2) v_number1 NUMBER(4):=1234;
(3) BEGIN
(4) DECLARE
(5)     v_number2 NUMBER(5,2):=543.21;
(6)     v_number1 NUMBER(5,2):=123.45;           -- 修改外部变量的值
(7) BEGIN
(8)     DECLARE
```

```
(9)      v_number3 NUMBER(6,3) :=123.456;
(10)     BEGIN
(11)     -- 执行代码3
(12)     END;
(13) -- 执行代码2
(14) END;
(15) -- 执行代码1
(16)END;
```

在这个PL/SQL代码段中，一共嵌套了3层DECLARE 和BEGIN部分，并在每一层的DECLARE部分中都定义一个变量。下面就来看一下这3个变量各自的作用域。

- ❑ 变量v\_number1：该变量定义在最外部的DECLARE块中，它的作用域是从代码的第1行一直到代码中的最后一个END结束，即代码的第16行。
- ❑ 变量v\_number2：该变量定义在内部的DECLARE块中，它的作用域是从代码的第5行一直到代码中的倒数第二个END结束，即代码的第14行。
- ❑ 变量v\_number3：该变量定义在最内层的DECLARE块中，它的作用域是从代码的第9行一直到代码中的倒数第三个END结束，即代码的第12行。

读者可能会注意到，在这个PL/SQL代码段中的第6行修改了外部变量v\_number1的值，这样做是可以的。在第二个BEGIN部分中，变量v\_number2和数据类型为NUMBER(5,2)的变量v\_number1都是可见的，可以在该执行部分中使用，那么数据类型为NUMBER(4)的变量v\_number1在第二个BEGIN部分是否可用呢？答案是肯定的。虽然数据类型为NUMBER(4)的变量v\_number1在第二个BEGIN部分是不可见的，但是从它的作用域可以知道，在第二个BEGIN部分是可以使用它的，既然是可以使用的，那么就会出现下一个问题，如何使用这个数据类型为NUMBER(4)的变量v\_number1呢？这就需要在外部的语句块中为其添加一个标记了。可以在PL/SQL的外部的语句块中添加一个outer1的标记，代码段如下：

```
<<outer1>>
(1)DECLARE
(2) v_number1 NUMBER(4):=1234;
(3)BEGIN
(4) DECLARE
(5)     v_number2 NUMBER(5,2):=543.21;
(6)     v_number1 NUMBER(5,2):=123.45;           -- 修改外部变量的值
(7) BEGIN
(8)     DECLARE
(9)         v_number3 NUMBER(6,3) :=123.456;
(10)        BEGIN
(11)        -- 执行代码3
(12)        END;
(13) -- 执行代码2
(14) END;
(15) -- 执行代码1
(16)END;
```

这段PL/SQL语句块中，在第1行的DECLARE部分的上面添加了一个标记outer1，此时如果希望在这个PL/SQL语句块的第二个BEGIN部分使用数据类型为NUMBER(4)的变量v\_number1，就可以使用outer1.v\_number1来引用这个变量。

## 18.9 数据类型之间的相互转换

在PL/SQL中，可以对不同的标量数据类型进行转换。标量数据类型之间的相互转换主要包括两种，一种是隐式的数据类型转换，一种是显式的数据类型转换。本节就来介绍PL/SQL中这两种数据类型之间的转换方式。

### 18.9.1 隐式数据类型转换

PL/SQL中，可以对数字和字符、字符和日期以及时间类型之间进行类型转换，这种转换都是由PL/SQL自动完成的，是隐式的数据类型转换。

例18.6 查询学生编号为s102203的学生t105这门课的学生成绩。

```
DECLARE
    v_result VARCHAR2(2);           -- 定义VARCHAR2类型的变量
BEGIN
    SELECT result
    INTO v_result
    FROM t_result
    WHERE stuID='s102203'
    AND curID= 't105';
    DBMS_OUTPUT.PUT_LINE ('学生成绩为'|| v_result ); -- 显示学生成绩
END;
```

在这段PL/SQL语句块中，DECLARE 部分定义的变量v\_result是VARCHAR2类型的，在BEGIN部分的SELECT语句中查询的课程成绩却是一个NUMBER(2)类型的变量，其结果如下所示。

学生成绩为：85

可以看到学生的成绩可以正常地显示。也就是说，PL/SQL中会将NUMBER类型的变量自动转换为VARCHAR2类型的数据。

但是有些类型转换在PL/SQL中却是不允许的。例如，在对字符数据进行转换时，不允许将一个数据类型为CHAR(5)的变量转换为VARCHAR2(2)，因为需要转换的数据类型VARCHAR2(2)中指定的字符长度比原来变量的数据类型CHAR(5)中指定的字符长度要小，不能保证有足够的空间对其进行转换。在对数值类型的数据进行转换时，也要考虑数据的精度和刻度范围。例如，将一个NUMBER(4,2)类型的数据转换为NUMBER(4)类型的数据时，程序运行时可能就会出现错误。

**说明** 虽然PL/SQL允许数据类型之间的隐式类型转换，但是如果需要对变量的数据类型进行转换，应该尽量使用显式的类型转换，以增强程序的可读性。

### 18.9.2 显式数据类型转换

PL/SQL中，也可以对数据类型进行显式的类型转换，这时需要用到SQL语句中内置的转换函数。Oracle中内置的转换函数主要包括TO\_CHAR()、TO\_DATE()、TO\_NUMBER()、ROWTOHEX()、HEXTORAWD()等。表18.2列出了SQL内置函数及这些函数可以转换的类型。



表18.2 SQL内置函数及这些函数可以转换的类型

SQL中的内置函数	可以转换的类型	SQL中的内置函数	可以转换的类型
TO_CHAR(number/date)	NUMBER、DATE	CHARTOROWID(char)	CHARACTER
TO_DATE(char)	CHARACTER	ROWIDTOCHAR(rowid)	ROWID
TO_NUMBER(char)	CHARACTER	TO_TIMESTAMP(char)	CHARACTER
ROWTOHEX(row)	ROW	TO_DAINTERVAL(char)	CHARACTER
HEXTORAWD(char)	CHARACTER	TO_YMINTERVAL(char)	CHARACTER

- ❑ TO\_CHAR()函数是将其指定参数转换为VARCHAR2类型的数据。
- ❑ TO\_DATE()函数是将其指定参数转换为DATE类型的数据。
- ❑ TO\_NUMBER()函数是将其指定参数转换为NUMBER类型的数据。
- ❑ ROWTOHEX()函数是将一个ROW数据转换为一个十六进制字符串。
- ❑ HEXTORAWD()函数是将一个十六进制的字符串转换为二进制的数字。该函数的参数char需要使用十六进制来表示。
- ❑ CHARTOROWID()函数将字符串值转换为RowID的数据类型表示。
- ❑ ROWIDTOCHAR()函数将RowID的数据类型转换为VARCHAR2类型的数据。
- ❑ TO\_TIMESTAMP()函数将指定参数转换为TIMESTAMP类型的数据（Oracle9i及其以上版本可用）。
- ❑ TO\_DAINTERVAL()函数将其指定参数转换为INTERVAL DAY TO MONTH类型的数据（Oracle9i及其以上版本可用）。
- ❑ TO\_YMINTERVAL()函数将其指定参数转换为INTERVAL YEAR TO MONTH类型的数据（Oracle9i及其以上版本可用）。

在18.9.1节中提到过，当需要对变量的数据类型进行转换时，应该尽量使用显式的类型转换，对于例18.6中的例子，如果使用显式的类型转换，可以使用如下的方法来完成。

```
DECLARE
    v_result VARCHAR2(2);
BEGIN
    SELECT TO_CHAR (result )
    INTO v_result
    FROM t_result
    WHERE stuID='s102203'
    AND curID= 't105';
    DBMS_OUTPUT.PUT_LINE ('学生成绩为'|| v_result );
END;
```

这段PL/SQL语句块中，在BEGIN部分的SELECT语句中使用了TO\_CHAR函数将列result中的数据显式转换为VARCHAR2类型的数据。这样做代码的可读性会更强。

18.10 小结

本章主要介绍PL/SQL中的一些基础知识，包括PL/SQL的块结构、PL/SQL的四个基本要素、变量的声明和使用、变量作用域、PL/SQL中的数据类型以及数据类型的转换等内容。这一部分内容也是以后深入学习PL/SQL的基础，了解了这些基础内容之后，在第19章将会介绍PL/SQL的流程控制语句。

## 第19章 PL/SQL中的控制结构

同其他的高级语言（例如，C、Java等）一样，PL/SQL也可以使用各种控制结构对PL/SQL的语句块进行控制。PL/SQL中的控制结构包括分支结构、循环结构和顺序结构。本章就来介绍PL/SQL中的这几种控制结构的使用方法。

本章重点：

- 分支控制结构
- 循环控制结构
- 顺序控制结构

### 19.1 分支控制

分支控制语句是通过判断条件表达式是TRUE或者FALSE来决定程序的执行。PL/SQL中分支控制语句包括IF语句和多分支的选择的CASE语句。IF语句又可以分为简单条件语句IF-THEN、二重条件分支语句IF-THEN-ELSE和多重条件分支语句IF-THEN-ELSEIF。这一节就来介绍PL/SQL中的IF语句的用法。多分支选择的CASE语句将在19.2节中介绍。

#### 19.1.1 IF-THEN简单条件语句

简单条件语句IF-THEN主要是用来对某一个单一条件进行判断。在使用IF-THEN简单条件语句时，如果满足IF中的指定条件，则会执行THEN语句后面的操作；如果不满足IF中的指定条件，则会退出分支条件语句。简单条件语句IF-THEN的语法规则如下：

```
IF expression THEN
    statement;
END IF;
```

在这个IF-THEN的条件语句中，expression是逻辑表达式或者关系表达式。如果expression的值为TRUE就会执行THEN后面的statement语句，否则，就会退出IF-THEN分支条件语句。关键字END IF是IF语句结束的标志。其执行的流程图如图19.1所示。

下面来看一个使用IF-THEN简单条件语句的例子。现在要查询教师信息表中教师编号为t156354的工资，如果该名教师的工资小于3000，那么就将该名教师的工资在原有的基础上加上300；如果工资大于3000，就将该名教师的工资在原有的基础上加上100。其PL/SQL代码如下：

```
DECLARE
```

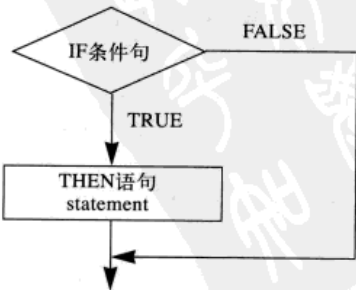


图19.1 IF-THEN简单条件流程图

```
v_salary NUMBER(6,2); -- 教师工资
v_teaID VARCHAR2(15) := ' t156354'; -- 定义表示教师编号的变量并为其赋初值
BEGIN
    /*查询指定教师编号的教师工资*/
    SELECT salary INTO v_salary
    FROM t_teacher
    WHERE teaID= v_teaID;
    /*判断教师工资，如果小于3000则执行更新操作*/
    IF v_salary < 3000 THEN
        UPDATE t_teacher SET salary = v_salary+300 -- 修改教师工资
        WHERE teaID =v_teaID;
    /*判断教师工资，如果大于3000则执行更新操作*/
    IF v_salary >3000 THEN
        UPDATE t_teacher SET salary = v_salary+100 -- 修改教师工资
        WHERE teaID =v_teaID;
    END IF;
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了两个变量，分别用来表示教师工资和指定教师编号，在BEGIN部分，首先使用SELECT语句查询指定教师编号的教师工资，然后使用IF-THEN语句对教师工资进行判断。如果该名教师的工资小于3000元，即第一个IF语句中的值为TRUE，那么就会执行第一个IF语句对应的THEN语句后面的UPDATE操作，将该名教师的工资在原有的基础上加上300；如果该名教师的工资大于3000元，即第二个IF语句中的值为TRUE，那么就会执行第二个IF语句对应的THEN语句后面的UPDATE操作，将该名教师的工资在原有的基础上加上100。

### 19.1.2 IF-THEN-ELSE条件分支语句

IF-THEN-ELSE条件分支语句可以根据条件判断来选择不同的操作。在使用IF-THEN-ELSE条件分支语句时，如果满足IF中的指定条件，则会执行THEN语句后面的操作；如果不满足IF中的指定条件，则会执行ELSE后面的操作。其语法规则如下：

```
IF expression THEN
    statement1;
ELSE
    statement2;
END IF;
```

在这个IF-THEN-ELSE的条件语句中，如果expression的值为TRUE，就会执行statement1语句；如果expression的值为FALSE，就会执行statement2的语句。关键字END IF是IF语句结束的标志。其执行的流程图如图19.2所示。

下面来看一个使用IF-THEN-ELSE的条件语句的例子。在18.6.3小节中介绍过PL/SQL嵌套表的使用。在当时讲解嵌套表的使用时曾经介绍过在创建完一个嵌套表之后，不能直接使用它，而是需要对其进行初始化操作，只有经过初始化操作之后，

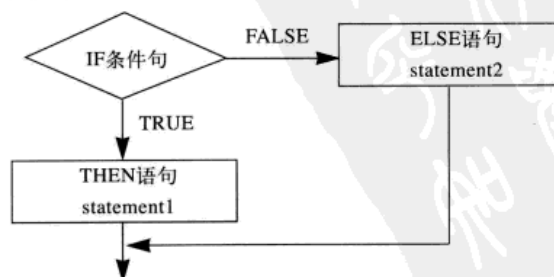


图19.2 IF-THEN-ELSE的条件分支流程图

才可以使用嵌套表。这是因为定义了PL/SQL嵌套表后，该表中虽然开始没有任何元素，但是同PL/SQL其他数据类型一样，PL/SQL嵌套表会被自动地赋上初始值NULL，此时当把一个元素添加到NULL的嵌套表时，系统就会提示上述的错误。当时并没有验证定义完一个嵌套表之后，嵌套表是否被自动地赋上初始值NULL。下面就使用IF-THEN-ELSE的条件语句来对此进行验证。

```
DECLARE
    TYPE char_table_type IS TABLE OF VARCHAR2(1)
    char_tab1 char_table_type;
    char_tab2 char_table_type := char_table_type();
    char_tab3 char_table_type := char_table_type('a','b','c');
BEGIN
    /*判断嵌套表char_tab1是否为NULL*/
    IF char_tab1 IS NULL THEN
        DBMS_OUTPUT.PUT_LINE (' char_tab1为NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' char_tab1不为NULL');
    End IF;
    /*判断嵌套表char_tab2是否为NULL*/
    IF char_tab2 IS NULL THEN
        DBMS_OUTPUT.PUT_LINE (' char_tab2为NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' char_tab2不为NULL');
    End IF;
    /*判断嵌套表char_tab3是否为NULL*/
    IF char_tab3 IS NULL THEN
        DBMS_OUTPUT.PUT_LINE (' char_tab3为NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' char_tab3不为NULL');
    End IF;
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了3个嵌套表。其中，嵌套表char\_tab1没有经过初始化操作，使用的是不带参数的构造方法对嵌套表char\_tab2进行初始化操作，使用的是带参数的构造方法对嵌套表char\_tab3进行初始化操作。在BEGIN部分使用IF-THEN-ELSE的条件语句对这三个嵌套表是否为NULL进行判断，如果嵌套表为NULL，则输出嵌套表为NULL的信息；如果嵌套表不为NULL，则输出嵌套表不为NULL的信息。其显示结果如下：

```
char_tab1为NULL
char_tab2不为NULL
char_tab3不为NULL
```

从显示的结果可以看到，如果不对嵌套表使用构造方法进行初始化操作，那么该嵌套表就会被赋予初始值NULL；如果对嵌套表使用构造方法进行初始化操作，那么嵌套表中就没有NULL的元素。这里需要说明的是，使用不带参数的构造方法对嵌套表进行初始化操作后，嵌套表中没有任何元素，可以认为是一个空表。

### 19.1.3 IF-THEN-ELSEIF多重条件分支语句

IF-THEN-ELSEIF多重条件分支语句主要是用于复杂条件的分支操作。在IF-THEN-ELSEIF多重条

件分支语句时，如果满足第一个IF中的指定条件，则会执行其对应的THEN语句后面的操作；如果不满足第一个IF中的指定条件，则会判断第一个ELSEIF后面的条件，如果满足其条件，就会执行其对应的THEN语句后面的操作，如果不满足其条件，就会检查第二个ELSEIF后面的条件（如果有的话），以此类推，直到遇到END IF结束整个条件分支语句的判断。IF-THEN-ELSEIF多重条件分支语句的语法规则如下：

```
IF expression1 THEN
    statement1;
ELSEIF expression2 THEN
    statement2;
ELSE
    statement3;
END IF;
```

在这个IF-THEN-ELSEIF多重条件分支语句中，如果expression1的值为TRUE，就会执行statement1语句；如果expression1的值为FALSE，就判断ELSEIF后面的表达式expression2，如果expression2的值为TRUE，则会执行statement2语句；如果expression2的值为FALSE，则会执行ELSE后面的statement3语句。关键字END IF是IF语句结束的标志。其执行的流程图如图19.3所示。

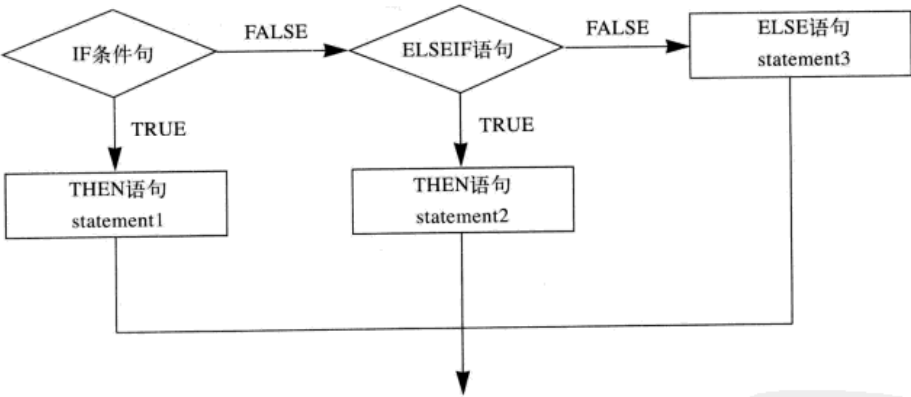


图19.3 IF-THEN-ELSEIF多重条件分支流程图

**注意** IF-THEN-ELSEIF多重条件分支语句中，ELSEIF是一个关键字，之间没有空格。关键字END IF之间有一个空格。

下面来看一个使用IF-THEN-ELSEIF多重条件分支语句的例子。现在需要在屏幕上输入一个指定教师的编号，并以该教师的教师编号作为限制条件查询教师所在的院系，根据该名教师所在的院系对其工资进行调整。如果教师所在的院系是计算机系，就将该名教师的工资在原有的基础上加上300；如果教师所在的院系是数学系，就将该名教师的工资在原有的基础上加上200；如果教师所在的院系是物理系，就将该名教师的工资在原有的基础上加上100。其PL/SQL代码如下：

```
DECLARE
    v_salary NUMBER(6,2);
    v_deptID VARCHAR2(15);
BEGIN
    -- 定义表示教师工资的变量
    -- 定义表示教师所在院系的变量
```



```
/*根据输入的教师编号查询教师工资和所在院系
SELECT salary , deptID INTO v_salary, v_deptID
FROM t_teacher
WHERE teaID= & teaID;
/*如果是计算机系，则将教师工资加上300*/
IF v_deptID =10 THEN
    UPDATE t_teacher SET salary = v_salary+300        -- 修改教师工资
    WHERE teaID =& teaID;
/*如果是数学系，则将教师工资加上200*/
ELSEIF v_deptID =15 THEN
    UPDATE t_teacher SET salary = v_salary+200        -- 修改教师工资
    WHERE teaID =& teaID;
/*如果是物理系，则将教师工资加上100*/
ELSE v_deptID =18 THEN
    UPDATE t_teacher SET salary = v_salary+100        -- 修改教师工资
    WHERE teaID =& teaID;
END IF;
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了两个变量，分别用来表示教师工资和教师所在院系，在BEGIN部分，使用SELECT语句根据输入的教师编号查询教师工资和教师所在院系，然后使用IF-THEN-ELSEIF多重条件分支语句对教师所在院系进行判断。如果字段deptID的值为10（即表示的是计算机系），则为计算机系中的每一个教师增加300元的工资；如果字段deptID的值为15（即表示的是数学系），则为数学系中的每一个教师增加200元的工资；如果字段deptID的值是18（即表示的是物理系），则为物理系中的每一个教师增加100元的工资。

#### 19.1.4 嵌套的IF语句

同其他的高级语言一样，PL/SQL中的IF语句也可以嵌套使用。以IF-THEN-ELSE条件分支语句为例，其嵌套的语法规则如下：

```
IF expression1 THEN
    statement1;
ELSE
    IF expression2 THEN
        statement2;
    ELSE
        IF expression3 THEN
            statement3;
        END IF;
    END IF;
END IF;
```

这里嵌套了三层IF-THEN-ELSE语句。该语句首先会对expression1的值进行判断，如果expression1的值为TRUE，则执行statement1；如果expression1的值为FALSE，则判断expression2的值；如果expression2的值为TRUE，则执行statement2；如果statement2的值为FALSE，则判断expression3的值；如果expression3的值为TRUE，则执行statement3；否则，则结束IF-THEN-ELSE语句，执行IF-THEN-ELSE语句后面的语句。

例如，可以将19.1.3节IF-THEN-ELSEIF多重条件分支语句中提到的例子使用嵌套的IF语句来实现。其实现代码如下：

```
DECLARE
    v_salary NUMBER(6,2);           -- 教师工资
    v_deptID VARCHAR2(15);          -- 教师所在院系
BEGIN
    /*根据输入的教师编号查询教师工资和所在院系*/
    SELECT salary , deptID INTO v_salary, v_deptID
    FROM t_teacher
    WHERE teaID= & teaID;
    /*如果是计算机系，则将教师工资加上300*/
    IF v_deptID =10 THEN
        UPDATE t_teacher SET salary = v_salary+300      -- 修改教师工资
        WHERE teaID =& teaID;
    ELSE
        /*如果是数学系，则将教师工资加上200*/
        IF v_deptID =15 THEN
            UPDATE t_teacher SET salary = v_salary+200  -- 修改教师工资
            WHERE teaID =& teaID;
        /*如果是物理系，则将教师工资加上100*/
        ELSE v_deptID =18 THEN
            UPDATE t_teacher SET salary = v_salary+100  -- 修改教师工资
            WHERE teaID =& teaID;
        END IF;
    END IF;
END;
```

上面的IF-THEN-ELSE嵌套语句理解起来可能不是很容易，而且可读性也比较差，通过对上面的IF-THEN-ELSE嵌套语句的执行过程的分析，可以很容易地使用IF-THEN-ELSEIF多重条件分支语句将其替换掉。替换IF-THEN-ELSE嵌套语句的IF-THEN-ELSEIF多重条件分支语句的语法规则如下：

```
IF expression1 THEN
    statement1;
ELSEIF expression2 THEN
    statement2;
ELSEIF expression3 THEN
    statement3;
```

这个IF-THEN-ELSEIF多重条件分支语句与本小节开头的那个IF-THEN-ELSE嵌套语句的执行逻辑是相同的，将这个程序代码与19.1.3小节中使用IF-THEN-ELSEIF多重条件分支语句实现的程序代码对比可以看到，使用IF-THEN-ELSEIF多重条件分支语句要比使用IF-THEN-ELSE嵌套语句的可读性更强，也更容易理解。所以，在实际应用中，如果可能的话，应该使用IF-THEN-ELSEIF多重条件分支语句替代IF-THEN-ELSE嵌套语句。

## 19.2 CASE语句

在Oracle9i之前，多重条件分支语句只能使用IF-THEN-ELSEIF语句来实现，在Oracle9i及其以后的

版本中可以使用CASE语句实现多重条件分支的操作。与IF-THEN-ELSEIF多重条件分支语句相比，CASE语句的书写更简洁、可读性更强、执行效率也更高，所以，在实际应用中，应该尽量把IF-THEN-ELSEIF语句改写成CASE语句。本节就来介绍多分支的选择CASE语句的用法。

### 19.2.1 实现等值比较的CASE语句

实现等值比较的CASE语句是以关键字CASE开头，关键字CASE后面需要指定一个表达式或者一个用于被检测的变量。CASE语句后面可以有一个或者多个WHEN子句，根据给定的表达式或者被检测变量的值决定哪个子句被执行。其语法规则如下：

```
CASE selector
  WHEN value1 THEN
    sequence_of_statements1;
  WHEN value2 THEN
    sequence_of_statements2;
  ...
  WHEN valueN THEN
    sequence_of_statementsN;
  [ELSE sequence_of_statementsN+1;]
END CASE ;
```

其中，selector用来指定一个表达式或者一个用于检测的变量；value1、value2一直到valueN表示需要比较的值；sequence\_of\_statements1、sequence\_of\_statements2一直到sequence\_of\_statementsN表示针对不同的比较值要执行不同的操作；如果所有的value值都不满足条件，则会执行ELSE后面的sequence\_of\_statementsN+1的语句。ELSE子句是可选的。关键字END CASE用来表示CASE语句的结束标志。

如果CASE语句中，selector的值与某一个WHEN子句的表达式值相同，就会执行该WHEN子句中的操作。例如，如果selector的值与value1的值相同，则CASE语句就会执行sequence\_of\_statements1的操作；同理，如果selector的值与value2的值相同，则CASE语句就会执行sequence\_of\_statements2的操作。当WHEN子句中的操作执行完毕之后，就会执行CASE语句后的下一个语句，后面的WHEN子句则不会被执行。如果selector的值与任何一个WHEN子句中的表达式值都不相同，则会执行ELSE子句中的操作。其执行的流程图如图19.4所示。

下面来看一个实现等值比较的CASE语句的例子。这里还以IF-THEN-ELSEIF多重条件分支语句提到的那个例子为例，看一下如何使用等值比较的CASE语句实现根据不同的院系编号为教师增加工资的功能。

```
DECLARE
  v_salary NUMBER(6,2);          -- 教师工资
  v_deptID VARCHAR2(15);         -- 教师所在院系
BEGIN
  /*根据输入的教师编号查询教师工资和所在院系*/
  SELECT salary , deptID INTO v_salary, v_deptID
  FROM t_teacher
  WHERE teaID= & teaID;
  CASE deptID
    /*如果是计算机系，则将教师工资加上300*/
```

```
WHEN 10 THEN
    UPDATE t_teacher SET salary = v_salary+300      -- 修改教师工资
    WHERE teaID =& teaID;
    /*如果是数学系，则将教师工资加上200*/
WHEN 15 THEN
    UPDATE t_teacher SET salary = v_salary+200      -- 修改教师工资
    WHERE teaID =& teaID;
    /*如果是物理系，则将教师工资加上100*/
WHEN 18 THEN
    UPDATE t_teacher SET salary = v_salary+100      -- 修改教师工资
    WHERE teaID =& teaID;
ELSE
    DBMS_OUTPUT.PUT_LINE (' 非计算机系、数学系或者物理系的教师');
END CASE ;
END;
```

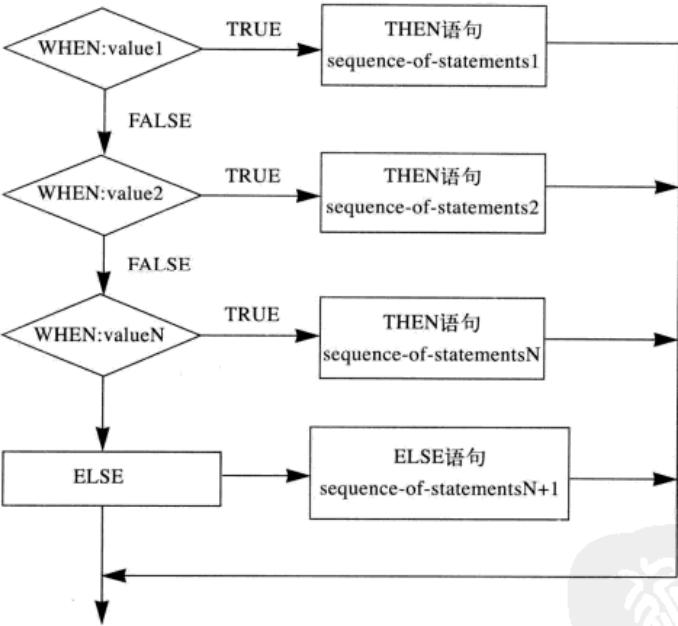


图19.4 CASE语句执行流程图

这段PL/SQL语句块中，在BEGIN部分使用CASE语句实现根据不同的院系编号为教师增加工资的功能。在CASE语句中，最后有一个ELSE子句，如果查询的教师不是计算机系、数学系和物理系的教师，就输出“非计算机系、数学系或者物理系的教师”的显示信息，并不对该名教师的工资进行修改操作。

可以看到，与IF-THEN-ELSEIF多重条件分支语句相比，使用CASE语句书写的代码要更简洁、可读性也更强，而且执行效率也要比使用IF-THEN-ELSEIF多重条件分支语句的高。

CASE语句中的ELSE子句是可选的。如果在CASE语句中不带ELSE子句，那么当CASE语句中被检测的变量或者表达式的值与任何一个WHEN子句指定的值都不匹配时，PL/SQL程序就会产生CASE\_NOT\_FOUND异常。例如，上面的例子中，如果在SELECT查询语句中根据输入的教师编号查询出该名

教师所在的院系不是计算机系、数学系或者物理系的，要是在CASE语句中没有最后的ELSE语句，那么此时程序会出现如下的错误。

```
ORA-06592 CASE not found while executing CASE statement.
```

**注意** 在PL/SQL中，如果要编写CASE语句，应该使用带有ELSE子句的CASE语句，以避免出现CASE\_NOT\_FOUND异常。

### 19.2.2 设定标记的CASE语句

同PL/SQL语句块一样，也可以使用“<<>>”为CASE语句添加标记。设定标记的CASE语句的语法规则如下：

```
[<<label_name>>]
CASE selector
  WHEN value1 THEN
    sequence_of_statements1;
  WHEN value2 THEN
    sequence_of_statements2;
  ...
  WHEN valueN THEN
    sequence_of_statementsN;
  [ELSE sequence_of_statementsN+1;]
END CASE [<<label_name>>];
```

其中，label\_name表示标记的名称，需要使用“<<>>”（双尖括号）将其括起来。当为CASE语句添加标记时，该标记必须出现在CASE语句的开头。如果为CASE语句添加了标记，那么该标记名称也可以出现在CASE语句的结尾处，但它是可选的。如果在END CASE处使用标记名称，那么该标记名称要和CASE语句开头设定的标记名称相匹配。

例如，对于19.2.1节实现等值比较的CASE语句中提到的那个例子，就可以为其添加相应的标记，其PL/SQL的代码如下：

```
DECLARE
  v_salary NUMBER(6,2);
  v_deptID VARCHAR2(15);
BEGIN
  /*根据输入的教师编号查询教师工资和所在院系*/
  SELECT salary , deptID INTO v_salary, v_deptID
  FROM t_teacher
  WHERE teaID= & teaID;
  <<teacher_addsalary >>
  CASE deptID
    /*如果是计算机系，则将教师工资加上300*/
    WHEN 10 THEN
      UPDATE t_teacher SET salary = v_salary+300
      WHERE teaID =& teaID;
    /*如果是数学系，则将教师工资加上200*/
    WHEN 15 THEN
```



```
UPDATE t_teacher SET salary = v_salary+200      -- 修改教师工资
WHERE teaID = & teaID;
/*如果是物理系，则将教师工资加上100*/
WHEN 18 THEN
UPDATE t_teacher SET salary = v_salary+100      -- 修改教师工资
WHERE teaID = & teaID;
ELSE
DBMS_OUTPUT.PUT_LINE (' 非计算机系、数学系或者物理系的教师');
END CASE teacher_addsalary ;
END;
```

### 19.2.3 搜寻式CASE语句

在CASE语句中，也可以不在关键字CASE之后使用selector用来指定一个表达式或者一个用于检测的变量，把这种CASE语句称为搜索式的CASE语句。其语法规则如下：

```
CASE
WHEN exception1 THEN
sequence_of_statements1;
WHEN exception2 THEN
sequence_of_statements2;
...
WHEN exceptionN THEN
sequence_of_statementsN;
[ELSE sequence_of_statementsN+1;]
END CASE ;
```

在搜寻式的CASE语句中，exception1、exception2...exceptionN表示布尔类型的表达式；sequence\_of\_statements1、sequence\_of\_statements2...sequence\_of\_statementsN表示满足表达式的值时要执行不同的操作。

在搜寻式的CASE语句中，如果某一个WHEN子句的布尔表达式为TRUE，则会执行该WHEN子句中对应的操作。例如，如果布尔表达式exception1的值为TRUE，则CASE语句就会执行sequence\_of\_statements1的操作；同理，如果布尔表达式exception2的值为TRUE，则CASE语句就会执行sequence\_of\_statements2的操作。当WHEN子句中的操作执行完毕之后，就会执行CASE语句后的下一个语句，后面的WHEN子句则不会被执行。如果WHEN子句中所有表达式的值都不为TRUE，则会执行ELSE子句中的操作。

下面来看一个使用搜寻式CASE语句的例子。这里以查询教师的年龄作为增加教师工资的条件。如果教师的年龄小于30岁，就将该名教师的工资在原有的基础上加上500；如果教师的年龄在30岁到40岁之间，就将该名教师的工资在原有的基础上加上300；如果教师的年龄在40岁到50岁之间，就将该名教师的工资在原有的基础上加上200；如果教师的年龄大于50，就将该名教师的工资在原有的基础上加上100。

```
DECLARE
v_salary NUMBER(6,2);      -- 教师工资
v_age VARCHAR2(15);        -- 教师年龄
BEGIN
/*根据输入的教师编号查询教师工资和年龄*/
SELECT salary , age INTO v_salary, v_age
```

```
FROM t_teacher
WHERE teaID= & teaID;
CASE
  /*如果教师的年龄小于30，则将教师工资加上500*/
  WHEN v_age <30 THEN
    UPDATE t_teacher SET salary = v_salary+500      -- 修改教师工资
    WHERE teaID =& teaID;
  /*如果教师的年龄小于40，则将教师工资加上300*/
  WHEN v_age <40 THEN
    UPDATE t_teacher SET salary = v_salary+300      -- 修改教师工资
    WHERE teaID =& teaID;
  /*如果教师的年龄小于50，则将教师工资加上200*/
  WHEN v_age <50 THEN
    UPDATE t_teacher SET salary = v_salary+200      -- 修改教师工资
    WHERE teaID =& teaID;
  ELSE
    /*如果教师的年龄大于50，则将教师工资加上100*/
    UPDATE t_teacher SET salary = v_salary+100      -- 修改教师工资
    WHERE teaID =& teaID;
END CASE;
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了两个变量，分别用来表示教师工资和教师年龄，在BEGIN部分，使用SELECT语句根据输入的教师编号查询教师工资和教师年龄，然后根据输入的教师编号对教师年龄进行判断。如果教师的年龄小于30，则将该名教师的工资在原有的基础上加上500；如果教师的年龄在30岁到40岁之间，则将该名教师的工资在原有的基础上加上300；如果教师的年龄在40岁到50岁之间，则将该名教师的工资在原有的基础上加上200；如果教师的年龄在50以上，则将该名教师的工资在原有的基础上加上100。

**注意** 在搜寻式的CASE语句中，WHEN子句中的表达式只能是布尔类型的表达式，不允许出现产生其他类型结果的表达式。

## 19.3 循环控制

循环控制语句与分支控制语句不同，它可以通过判断给定的条件重复执行某一个语句或者是多条语句的代码段。PL/SQL中循环控制语句主要包括LOOP循环语句、WHILE -LOOP循环语句和FOR-LOOP循环语句。本节就来介绍这三种循环语句的使用方法。

### 19.3.1 LOOP循环语句

LOOP循环是PL/SQL中简单的循环语句。在LOOP循环中，执行循环的语句被放到关键字LOOP和END LOOP之间。其语法规则如下：

```
LOOP
  statement;
  ...
```

```
EXIT [WHEN condition]
END LOOP;
```

其中，statement是要执行的循环语句，这样的循环语句可以有一条，也可以有多条。在LOOP循环语句中，如果有多条statement语句，则这些语句会被顺序地执行。EXIT语句用来终止LOOP循环。当程序遇到EXIT语句时，循环就会立即停止，程序会跳出循环体，转而执行LOOP循环后面的语句。在EXIT关键字之后，还有一个WHEN子句，可以根据condition的条件跳出循环，当condition的值为TRUE时，就会终止LOOP循环，转而执行循环后面的语句。这个WHEN子句是可选的。

下面来看一个使用LOOP循环语句的例子。使用LOOP循环语句计算1~10的加和，并将结果输出。

```
DECLARE
    v_count INT:=1;
    v_sum INT:=0;
BEGIN
    LOOP
        v_sum = v_sum+1;
        EXIT WHEN v_count =10;
        v_count:=v_count+1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE (' v_sum = ' || v_sum);
END;
```

-- 定义表示计数器的变量  
-- 定义表示1到10加和的变量  
-- 计算1到10的加和  
-- 终止循环的条件  
-- 计数变量加1  
-- 显示加和信息

这段PL/SQL语句块是使用LOOP循环语句计算1~10的加和。在DECLARE部分定义了两个变量，分别用来表示计数器和1~10的加和。在BEGIN部分，使用了LOOP循环语句，其中EXIT WHEN v\_count =10表示终止LOOP循环的条件。即当计数器的值变成10以后，就要跳出循环体，不再执行循环体中的语句了。最后使用DBMS\_OUTPUT.PUT\_LINE将1到10的加和信息显示处理。其显示结果如下所示。

```
v_sum =55
```

**注意** LOOP循环语句中循环体的statement语句会执行一次。

也可以使用简单的IF语句替代EXIT-WHEN语句。例如上面的例子中，EXIT-WHEN语句就可以使用IF-THEN语句代替。上面的例子如果使用IF-THEN语句代替EXIT-WHEN语句，可以写成如下的形式。

```
IF count =10 THEN
    EXIT;
END IF; -- 相当于EXIT WHEN count = 10;
```

在LOOP循环中使用IF语句替代EXIT-WHEN语句在逻辑上是等价的，但是使用EXIT-WHEN语句的可读性要更强，也有利于对程序的理解。

同PL/SQL语句块一样，也可以使用“<<>>”为LOOP循环语句添加标记。为LOOP循环语句添加标记的语法规则如下：

```
<<loop_name>>
LOOP
    statement;
```

## 零基础学SQL

```
...  
EXIT [WHEN condition]  
END LOOP[<<loop_name>>];
```

其中，loop\_name表示标记的名称，需要使用“<<>>”（双尖括号）将其括起来。当为LOOP循环语句添加标记时，该标记必须出现在LOOP循环语句的开头。如果为LOOP循环语句添加了标记，那么该标记名称也可以出现在LOOP循环语句的结尾处，但它是可选的。如果在END LOOP处使用标记名称，那么该标记名称要和LOOP循环语句开头设定的标记名称相匹配。如果要为上面的LOOP循环语句添加标记，其PL/SQL 代码如下：

```
DECLARE  
    v_count INT:=1;           -- 定义表示计数器的变量  
    v_sum INT:=0;             -- 定义表示1~10加和的变量  
BEGIN  
    <<loop_sum >>  
    LOOP  
        v_sum = v_sum +1;      -- 计算1~10的加和  
        EXIT WHEN v_count =10; -- 终止循环的条件  
        v_count:=v_count+1;    -- 计数变量加1  
    END LOOP loop_sum;  
    DBMS_OUTPUT.PUT_LINE (' v_sum = ' || v_sum); -- 显示加和信息  
END;
```

### 19.3.2 WHILE -LOOP循环语句

WHILE-LOOP循环语句是根据WHILE条件给定的布尔表达式的结果是TRUE还是FALSE，来决定是否需要重复执行循环体中的语句。其语法规则如下：

```
WHILE condition LOOP  
    statement;  
    ...  
END LOOP;
```

其中，condition是指定的布尔表达式；WHILE-LOOP循环语句会根据该布尔表达式的值来决定是否需要重复执行循环体中的语句。如果condition的值为TRUE，则WHILE-LOOP循环语句中的statement语句就会被执行；如果condition的值为FALSE，则会退出WHILE-LOOP循环语句，转而执行WHILE-LOOP循环语句后面的语句。其执行的流程图如图19.5所示。

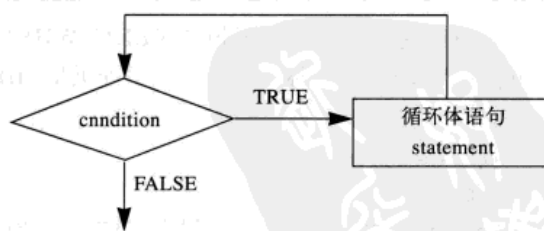


图19.5 WHILE-LOOP循环流程图

下面来看一个使用WHILE-LOOP循环语句的例子。使用WHILE-LOOP循环语句计算1到10的加和，并将结果输出。

```
DECLARE  
    v_count INT:=1;           -- 定义表示计数器的变量  
    v_sum INT:=0;             -- 定义表示1到10加和的变量  
BEGIN
```

```
WHILE v_count<10 LOOP
    v_sum = v_sum + 1;           -- 计算1到10的加和
    v_count:=v_count+1;        -- 计数变量加1
END LOOP;
DBMS_OUTPUT.PUT_LINE (' v_sum = ' || v_sum);  -- 显示加和信息
END;
```

这段PL/SQL语句块是使用LOOP循环语句计算1到10的加和。在DECLARE部分定义了两个变量，分别用来表示计数器和1到10的加和。在BEGIN部分，使用了WHILE-LOOP循环语句，其中v\_count<10表示终止WHILE-LOOP循环的条件。即当计数器的值变成10以后，就要跳出循环体，不再执行循环体中的语句了。最后使用DBMS\_OUTPUT.PUT\_LINE将1到10的加和信息显示处理。其显示结果如下所示。

```
v_sum =55
```

由于在WHILE-LOOP循环语句中，条件是在循环开始检测的，所以WHILE-LOOP循环体中的语句可能一次都没有执行。对于上面的例子，读者可以考虑一下，什么情况下，WHILE-LOOP循环体中的语句一次也不会被执行。

### 19.3.3 FOR-LOOP循环语句

在LOOP循环和WHILE -LOOP循环语句中，循环的次数是未知的，但是在有些情况下，循环的次数是可以预知的，例如，计算1+2+3+...+10的值，这里很明显是需要循环10次，像这样可以明确地知道循环次数的问题，就可以使用FOR-LOOP循环语句来完成。其语法规则如下：

```
FOR counter IN [REVERSE]lower_bound..higher_bound LOOP
    statement;
    ...
END LOOP;
```

其中，counter表示的是隐式声明的循环控制变量；lower\_bound用来指定counter的下限；higher\_bound用来指定counter的上限；statement表示循环体中的语句；在lower\_bound和higher\_bound之间有“..”（两个“点”），这两个“点”是作为FOR-LOOP循环语句的范围操作符来使用的。

在FOR-LOOP循环语句中还有一个REVERSE关键字。该关键字表示循环控制变量将从上限值开始，依次减1，执行循环体内的语句，该关键字是可选的。如果不使用REVERSE关键字，则FOR-LOOP循环语句将从指定的循环控制变量的下限值开始，依次加1，执行循环体中的语句。默认情况下，FOR-LOOP循环语句中的循环总是从循环控制变量的下限开始，依次加1到上限。

下面来看一个使用FOR-LOOP循环语句的例子，在18.6.3节中介绍了嵌套表的定义和使用方法。PL/SQL中嵌套表类似于C语言中的数组。如果在嵌套表中存储了数据，那么就可以使用FOR-LOOP循环语句将其他的数据一一输出。

```
DECLARE
    TYPE char_table_type IS TABLE OF VARCHAR2(1)
    char_tab char_table_type := char_table_type('a','b','c','d','e') ;-- 初始化嵌套表
BEGIN
    FOR v_count IN 1...5 LOOP
        DBMS_OUTPUT.PUT_LINE (char_tab(v_count)|| ' ');  -- 显示嵌套表中元素的信息
    END LOOP;
END;
```



## 零基础学SQL

这段PL/SQL语句块中，在DECLARE部分定义了一个可以容纳字符类型的嵌套表char\_tab，并使用构造方法对其进行了初始化操作，此时在该嵌套表中存储了5个字符数据。在BEGIN部分使用FOR循环将这5个数据一一输出显示。其查询结果如下：

a b c d e

**注意** 在FOR-LOOP循环语句中，循环控制变量counter只能作为常量引用，不允许在程序中为其赋值。如果lower\_bound的值与higher\_bound的值相等，则FOR-LOOP循环语句中循环体的语句只会被执行一次。

### 19.3.4 嵌套循环

同其他的高级语言（例如，C、Java等）一样，PL/SQL中的循环语句也可以嵌套使用。例如，在LOOP嵌套循环语句中，可以使用EXIT语句结束任意一个LOOP循环，只要在LOOP循环中添加一个标记，然后在EXIT语句使用这个标记就可以退出该标记指定下的LOOP循环。

```
BEGIN
<<outer>>                                -- 外层循环标记
  LOOP
    ...
    LOOP
      ...
      EXIT outer WHEN ...                -- 退出外层循环
    END LOOP;
    ...
  END LOOP outer;
END;
```

## 19.4 顺序控制

在PL/SQL中，不仅有分支控制语句和循环控制语句，还有顺序控制语句。顺序控制语句主要包括GOTO语句和NULL语句。GOTO语句主要是用来简化程序逻辑，NULL语句主要是用来提高PL/SQL程序的可读性。在PL/SQL中顺序控制语句并不是经常用到。这一节就来介绍使用GOTO语句和NULL语句的方法。

### 19.4.1 GOTO语句

GOTO语句主要用于跳转到指定的标记处，执行指定标记处的代码。使用GOTO语句跳转到标记处的语法规则如下。

```
GOTO label_name;
```

其中，label\_name表示指定的标记。在PL/SQL程序中，标记名称是使用“<<>>”（双尖括号）括起来的。程序在执行的时候，当遇到GOTO语句时，就会将控制权交给指定的标记处的语句或者语句块。GOTO语句既可以向上跳转，也可以向下跳转。

**注意** 在使用GOTO语句执行跳转时，GOTO语句后面指定的标记中至少应该含有一条可执行的语句。

下面来看一个使用GOTO语句跳转的例子。使用LOOP循环语句计算1到10的加和，并在LOOP循环语句中使用GOTO语句跳转的方式结束循环体的语句，将结果输出。

```
DECLARE
    v_count INT:=1;           -- 定义表示计数器的变量
    v_sum   INT:=0;           -- 定义表示1到10加和的变量
BEGIN
    LOOP
        v_sum = v_sum+1;       -- 计算1到10的加和
        IF count =10 THEN
            GOTO quit;         -- 跳出循环
        END IF;
        v_count:=v_count+1;    -- 计数变量加1
    END LOOP;
    << quit >>                -- quit标记
    DBMS_OUTPUT.PUT_LINE (' v_sum = ' || v_sum); -- 显示加和信息
END;
```

这段PL/SQL语句块是计算1到10的加和，这里在IF-THEN语句中使用GOTO语句跳出LOOP循环，并执行quit标记处的语句，这里执行的是将1到10的加和信息显示出来。

使用GOTO语句除了可以向下跳转之外，也可以实现向上跳转。例如下面这个PL/SQL代码片段就是使用GOTO语句实现向上跳转的例子。

```
BEGIN
    ...
    <<insert _row>>           -- 指定标记
    BEGIN
        INSERT INTO t_student ....
        ...
    END;
    ...
    GOTO insert _row;         -- 跳转到指定标记
    ...
END;
```

这个PL/SQL的代码片段中有两个BEGIN部分，并为内层的BEGIN部分中设定了一个insert \_row标记，用来执行相应的插入操作。在外层的BEGIN部分中有一个GOTO语句，这个GOTO语句跳转到了指定标记insert \_row处，并执行指定标记处的代码。

PL/SQL中对GOTO语句做了一些限制。有些情况下，使用GOTO语句执行跳转操作是非法的。例如，不能使用GOTO语句跳转到IF语句中，因为如果允许GOTO语句跳转到IF语句中，那么即使IF语句中的判断条件不为TRUE，IF语句中的程序也可以被执行，那样是不符合逻辑的。同样道理，也不能使用GOTO语句跳转到CASE语句、循环语句中。下面是一些非法使用GOTO语句的介绍。

❑ 使用GOTO语句跳转到IF语句、CASE语句、LOOP循环语句或者内层的语句块中是不允许的。

```
BEGIN
    ...
```

```
GOTO insert_row;                -- 不能跳转到IF语句中
...
IF deptID =10 THEN
...
  << insert_row>>
  INSERT INTO ...
END IF;
END;
```

❑ 使用GOTO语句从一个IF语句的一个分支跳转到另一个分支是非法的。

```
BEGIN
...
IF deptID =10 THEN
...
  GOTO insert_row;                -- 不能跳转到一个IF语句中的ELSE分支中
ELSE
...
  << insert_row>>
  INSERT INTO t_student ...
END IF;
END;
```

❑ 使用GOTO语句从CASE语句中的一个WHEN子句跳转到另一个WHEN子句是非法的。

```
BEGIN
...
CASE deptID
  WHEN 10 THEN
...
    GOTO update_row;                -- 不能从CASE语句的一个WHEN子句跳转到另一个WHEN子句
  WHEN 15 THEN
    <<update_row>>
    UPDATE t_teacher SET ...
  WHEN 18 THEN
...
  ELSE
...
END CASE ;
END;
```

❑ 使用GOTO语句跳转到一个指定的标记处，但是在该标记处并没有可执行的语句，这种情况是不允许的。

```
BEGIN
  LOOP
    v_sum = v_sum+1;                -- 计算1到10的加和
    IF count =10 THEN
      GOTO quit;                    -- 不能跳转到没有可执行语句的标记处
    END IF;
    v_count:=v_count+1;              -- 计数变量加1
  END LOOP;
```

```
<< quit >>                                -- quit 标记
END;
```

❑ 使用GOTO语句从一个封闭的语句块跳入它的子块是非法的。

```
BEGIN
...
IF deptID = 10 THEN
    GOTO delete_row;                        -- 不能从一个封闭的语句块跳入它的子块
END IF;
...
BEGIN
...
<<delete_row >>
DELETE FROM t_stuent
WHERE ...
END;
END;
```

❑ 使用GOTO语句从一个异常控制语句块中跳转到当前块是非法的。

```
BEGIN
...
<<insert_row>>
INSERT INTO t_student
EXCEPTION
WHEN ZERO_DIVIDE THEN
...
GOTO insert_row;                          -- 不能从一个异常控制语句块中跳转到当前块
END;
```

GOTO语句虽然可以方便地进行跳转，但是滥用GOTO语句的后果也是很严重的。它可能会使程序结构变得混乱，程序也不容易理解和维护。所以，使用GOTO语句一定要特别小心。如果通过其他方式可以实现GOTO语句的跳转功能就不应该使用GOTO语句来完成。

很多情况下，通过PL/SQL的控制结构语句也可以实现GOTO语句跳转的功能，也可以使用异常处理语句来完成深度嵌套的跳转。例如，要从一个深层嵌套中跳转到一个异常控制块，就可以通过抛出一个异常来实现，而不要使用GOTO语句来实现。

**注意** 在实际应用中，不建议使用GOTO 语句。因为GOTO 语句会降低程序的可读性，使程序变得更复杂，不容易理解，也不利于代码的调试和维护。

### 19.4.2 NULL语句

NULL语句不执行任何操作，而是简单地把控制权交给下一条执行语句。使用NULL语句可以提高程序的可读性。例如，可以使用NULL满足IF语句中的语法，而NULL语句本身没有任何操作。

例如，对于19.1.3小节IF-THEN-ELSEIF多重条件分支语句中提到的例子，根据输入的教师编号所在院系对教师工资进行修改，如果不是计算机系、数学系或者物理系的教师，则不对教师工资进行修改操作。

## 零基础学SQL

```
DECLARE
    v_salary NUMBER(6,2);
    v_deptID VARCHAR2(15);
BEGIN
    /*根据输入的教师编号查询教师工资和所在院系*/
    SELECT salary , deptID INTO v_salary, v_deptID
    FROM t_teacher
    WHERE teaID= & teaID;
    /*如果是计算机系，则将教师工资加上300*/
    IF v_deptID =10 THEN
        UPDATE t_teacher SET salary = v_salary+300 -- 修改教师工资
        WHERE teaID =& teaID;
    /*如果是数学系，则将教师工资加上200*/
    ELSEIF v_deptID =15 THEN
        UPDATE t_teacher SET salary = v_salary+200 -- 修改教师工资
        WHERE teaID =& teaID;
    /*如果是物理系，则将教师工资加上100*/
    ELSEIF v_deptID =18 THEN
        UPDATE t_teacher SET salary = v_salary+100 -- 修改教师工资
        WHERE teaID =& teaID;
    ELSE
        /* 如果是非计算机系、数学系或者是物理系的教师，则不执行任何操作*/
        NULL;
    END IF;
END;
```

在19.4.1节中介绍使用GOTO语句的限制的时候，提到过如果使用GOTO语句跳转到一个指定的标记处，但是在该标记处并没有可执行的语句，这种情况是不允许的。例如下面这个例子。

```
DECLARE
    v_count INT:=1;
    v_sum INT:=0;
BEGIN
    LOOP
        v_sum = v_sum+1;
        IF count =10 THEN
            GOTO quit;
        END IF;
        v_count:=v_count+1;
    END LOOP;
    << quit >>
END;
```

-- 定义表示计数器的变量  
-- 定义表示1到10加和的变量  
-- 计算1到10的加和  
-- 不能跳转到没有可执行语句的标记处  
-- 计数变量加1  
-- quit标记

这段PL/SQL语句块是计算1到10的加和，这里在IF-THEN语句中使用GOTO语句跳出LOOP循环，但是从这段程序中可以看到，GOTO 语句中指定的quit标记处并没有可执行的代码，因此这种跳转方式是非法的。

如果在上面的这个程序中希望跳转到quit的标记处，也可以实现。其实现的方法就是在quit的标记的后面添加一个NULL语句。

```
DECLARE
```



```
v_count INT:=1;          -- 定义表示计数器的变量
v_sum INT:=0;            -- 定义表示1到10加和的变量
BEGIN
  LOOP
    v_sum = v_sum+1;      -- 计算1到10的加和
    IF count =10 THEN
      GOTO quitloop;      -- 跳出循环
    END IF;
    v_count:=v_count+1;   -- 计数变量加1
  END LOOP;
  << quit >>             -- quit标记
  NULL
END;
```

这段PL/SQL语句块中，在quit标记处添加了一个NULL语句，该NULL语句并不执行任何操作，此时使用GOTO语句跳转到quit标记处就没有问题了。

## 19.5 小结

本章主要介绍了PL/SQL语句块中控制结构的使用方法。PL/SQL语句块中控制结构主要包括分支控制结构、循环控制结构和顺序控制结构。通过本章的介绍，需要掌握IF语句、CASE语句、LOOP循环语句、WHILE-LOOP循环语句和FOR-LOOP循环语句的使用方法。PL/SQL中，IF语句和循环语句是允许嵌套的。在实际应用中，主要使用的是分支控制结构和循环控制结构。顺序控制结构中的GOTO语句一般是不建议使用的。



## 第20章 使用游标

在PL/SQL中可以使用游标处理数据。通过使用游标可以大大提高PL/SQL程序对数据处理的能力。在Oracle 9i及其以后的版本中，还增加了使用BULK COLLECT子句批量绑定数据和使用CURSOR表达式实现嵌套游标的功能。本章将主要介绍如何使用显式游标进行多行数据的查询、游标FOR循环以及游标变量的使用，另外还将介绍游标属性以及嵌套游标的使用等内容。

本章重点：

- ☐ 游标的概念
- ☐ 显式游标的使用方法
- ☐ 游标属性以及显式游标和隐式游标属性之间的比较
- ☐ 游标循环的使用
- ☐ 游标变量的使用
- ☐ 嵌套游标的使用

### 20.1 什么是游标

在Oracle中，使用SQL语句（例如，SELECT、INSERT、UPDATE、DELETE等）进行查询、修改、插入、删除等操作时，数据库管理系统会在内存中为其分配一个区域，这个区域是一段上下文的缓冲区。在这段区域中包含了SQL语句处理过程的必要信息。游标就是指向该段上下文缓冲区中数据的指针。

在PL/SQL中使用游标可以方便地控制上下文缓冲区以及语句处理过程中数据的变化。如果希望对结果集中的数据进行处理，就需要声明一个指向这个结果集的游标。通过使用游标，不仅可以查询数据库中的记录，对查询的结果集中的每一行记录执行不同的操作，也可以基于游标的位置对数据表中的记录进行更新操作（例如，修改数据、删除数据等）。

游标有两种类型，一种是隐式游标，一种是显式游标。对所有的SQL数据操作语句，PL/SQL都会为其声明一个隐式游标。使用游标可以从含有多条记录的结果集中提取一条数据记录。当然，也可以使用游标从结果集中返回多条数据记录。如果希望从结果集中返回多条数据记录，就需要声明一个显式游标，并通过游标FOR循环或者使用BULK COLLECT子句批量绑定数据的方式得到多条数据记录。

### 20.2 使用显式游标

在使用SQL查询语句进行查询时，其返回的结果集可能有零行，可能有一行，也可能有多行。如果SELECT语句返回的是一个含有多行数据记录的结果集，就可以通过声明一个显式游标来处理结果集中的每一行数据。也可以使用BULK COLLECT子句批量处理多行数据。使用显式游标需要包括4个过程：

声明游标、打开游标、提取数据和关闭游标。这一节就以上述4个过程为基础讲解使用显式游标处理数据的方法。

### 20.2.1 声明游标

要想使用游标提取数据，首先需要声明一个游标。声明游标时，需要为游标定义一个名字，并为其指定一个对应的SELECT语句。声明游标的语法规则如下：

```
CURSOR cursor_name IS select_statement;
```

其中，cursor\_name表示游标的名字；select\_statement是一个与cursor\_name游标对应的查询语句。这里的SELECT查询语句中不能含有INTO子句。游标的声明可以放到PL/SQL语句块、子程序或者包的声明部分。

**注意** 在引用游标之前，必须使用声明游标的语句对其进行声明。如果在引用之前没有对游标进行声明，则这样的引用就是非法的。

下面来看一个声明游标的例子。

```
DECLARE
    CURSOR c_result IS
        SELECT R.stuID,C.curID, C.curName,R.result
        FROM t_result R,T_curriculum C
        WHERE R.curID=C.curID
        ORDER BY R.stuID
```

-- 声明游标

这段PL/SQL代码片段中，在DECLARE部分声明了一个游标。其中，c\_result表示游标的名字，IS关键字后面的SELECT语句是查询学生所选课程的成绩信息。通过这样一个游标声明，就把c\_result游标与查询学生所选课程的成绩信息的SELECT语句相互关联了起来。

游标声明也可以引用PL/SQL的变量。如果在声明游标时，其对应的SELECT语句中引用了变量，则这些变量被称为绑定变量。例如，下面的游标声明也是合法的。

```
DECLARE
    v_stuID      t_student . stuID %TYPE;
    CURSOR cursor_student IS
        SELECT stuName, age, sex
        FROM t_student
        WHERE stuID = v_stuID;
```

-- 学生编号  
-- 声明游标

这段PL/SQL代码片段中，在DECLARE部分，定义了一个表示学生编号的变量，在声明游标的语句中使用了这个变量。这里的SQL语句是用来查询学生的基本信息。

这里需要说明的一点是，游标的名字是一个未声明的标识符，而不是PL/SQL中的变量名。因此，不能把值赋给一个游标名，也不能在一个表达式中使用游标。但是，游标和变量作用域规则是相同的（有关变量的作用域可以参看18.8节）。

游标也是可以接收参数的，但是游标参数的值只在打开游标的时候才可以被使用。有关游标接受参数的内容可以参看20.4.5小节。

## 20.2.2 打开游标

在声明完一个游标之后，就可以使用OPEN语句打开这个游标了。只有在打开了游标之后，才可以执行相关数据提取操作。打开游标的语法规则如下：

```
OPEN cursor_name;
```

其中，cursor\_name表示前面已经声明的游标的名字。当游标打开之后，就可以执行其对应的SELECT语句的操作，并可以将数据查询的结果放入到游标的缓冲区中。使用OPEN语句打开一个游标，其对应的SELECT语句中的结果集中的行并没有被选取，如果想使游标取得数据，需要使用FETCH语句。

下面来看一个打开游标的例子。

```
DECLARE
    v_stuID      t_student . stuID %TYPE;      -- 学生编号
    v_stuName     t_student . stuName%TYPE;     -- 学生姓名
    v_age         t_student . age%TYPE;         -- 学生年龄
    v_sex         t_student . sex%TYPE E;       -- 学生性别
    CURSOR cursor_student IS
        SELECT stuName, age, sex
        FROM t_student
        WHERE stuID = v_stuID;
BEGIN
    v_stuID := 's102203';                      -- 变量赋值
    OPEN cursor_student;                       -- 打开游标
    ...
END;
```

这段PL/SQL代码片段中，在BEGIN部分，为表示学生编号的变量v\_stuID赋值，并使用OPEN语句打开了一个游标。如果使用OPEN语句打开了游标，那么它就不能再次打开，要想再次打开该游标，只能把它关闭之后再打开。

当打开一个游标时，会创建一个执行该结果集的指针。绑定变量只有在游标被打开时，其值才会被计算。例如下面的这个例子。

```
DECLARE
    v_stuID      t_student . stuID %TYPE;      -- 学生编号
    v_stuName     t_student . stuName%TYPE;     -- 学生姓名
    v_age         t_student . age%TYPE;         -- 学生年龄
    v_sex         t_student . sex%TYPE E;       -- 学生性别
    CURSOR cursor_student IS
        SELECT stuName, age, sex
        FROM t_student
        WHERE stuID = v_stuID;
BEGIN
    v_stuID := 's102203';                      -- 变量赋值
    OPEN cursor_student;                       -- 打开游标
    ...
    v_stuID := 's206363';                      -- 修改v_stuID变量
END;
```

这段PL/SQL代码片段中，在BEGIN部分，首先为表示学生编号的变量赋值为s102203，之后打开了游标cursor\_student，此时查询的结果集中就是学生编号为s102203对应的学生信息，即使学生编号的值在打开游标之后发生了变化，其查询的结果集也不会变化，还是学生编号为s102203对应的学生信息。如果想得到修改后的学生编号对应的学生信息，必须先将游标关闭，然后再将其打开，此时看到的还是学生编号修改后对应的学生信息。

如果对一个已经打开的游标再使用OPEN语句，Oracle数据库管理系统就会给出下面的一个错误提示。

```
ORA-06511:PL/SQL:cursor already open
```

### 20.2.3 从游标中取得结果

在完成了声明游标和打开游标的操作之后，就可以使用游标检索结果集中的数据了。使用游标检索结果集中的数据是使用FETCH语句来完成。其语法规则如下：

```
FETCH cursor_name INTO variable1 [ variable2...]
```

其中，cursor\_name为游标的名字，该游标需要是已经在前面声明并打开的游标。variable1、variable2表示的是声明的变量。这里的变量可以是一个标量变量，也可以是一个记录或者PL/SQL中的集合变量。多个变量之间需要用逗号分隔。

**注意** FETCH语句中需要包含INTO子句。而声明游标时，select\_statement查询语句不能包含INTO子句。

下面来看一个使用FETCH语句的例子。例如，如果要使用FETCH语句提取学生信息表中的数据记录，可以使用下面的方法完成。

```
FETCH cursor_student INTO v_stuName,v_age,v_sex
```

这里的cursor\_student是在前面已经声明并打开的游标，v\_stuName、v\_age、v\_sex分别是游标接收结果集中数据的变量。

在使用FETCH语句提取数据记录时，其INTO子句中的变量列表中的变量必须与游标查询返回的列值的数据类型相兼容。如果INTO列表中变量的数据类型与游标查询返回的列值的数据类型不互相兼容，则Oracle数据库管理系统会给出下面的一个错误提示。

```
PLS-394:wrong number of values
```

使用FETCH语句提取数据，可以使用一个简单的LOOP循环方式（可以参看20.4.1节）。FETCH语句每次只能从结果集中提取一条记录，如果希望在FETCH语句中取得多条记录，可以使用BULK COLLECT子句。

在使用FETCH语句提取数据记录时，它每次只能从结果集中提取一条记录，同时将游标下移，指向当前记录的下一条记录，以此类推。因此当检索结果集最后一次执行FETCH语句时，会取不到数据，数据库管理系统也不会产生异常，此时就需要使用游标的%FOUND或者%NOTFOUND属性（游标游标属性的内容可以参看20.3节）。

### 20.2.4 关闭游标

当查询的结果集检索完成之后，就可以关闭游标了。关闭游标是使用CLOSE语句来完成的。关闭



游标后，与游标相关的资源将会被释放。关闭游标的语法规则如下：

```
CLOSE cursor_name;
```

其中，cursor\_name表示关闭游标的名字。这里的cursor\_name是前面已经打开的游标名。游标被关闭后还可以重新将其打开。

如果一个游标关闭之后，还要使用它来检索数据，那么Oracle数据库管理系统会给出下面的一个错误提示。

```
ORA-1001:Invalid Cursor
```

同样，如果要对一个已经关闭了的游标进行操作也是非法的，此时Oracle数据库管理系统会抛出INVALID\_CURSOR异常。

### 20.2.5 使用BULK COLLECT子句批量绑定数据

FETCH语句每次只能从结果集中提取一条记录，Oracle 9i及其以后的版本中，如果希望在FETCH语句中取得多条记录，可以使用BULK COLLECT子句。BULK COLLECT子句可以批量绑定数据，将结果集中的所有行一次性地都提取出来。

下面来看一个使用BULK COLLECT子句批量绑定数据的例子。这个例子中，将计算机系（院系编号为t\_10）的教师的信息全部输出。

```
DECLARE
    TYPE teacher_table_type IS TABLE OF t_teacher %ROWTYPE;
    teacher_table teacher_table_type;
    CURSOR cursor_teacher IS                -- 声明游标
        SELECT teaName,,dept
        FROM t_teacher,
        WHERE deptID = 't_10';
BEGIN
    OPEN cursor_teacher;                    -- 打开游标
    FETCH cursor_teacher
    BULK COLLECT INTO teacher_table;        -- 使用BULK COLLECT子句提取数据
    CLOSE cursor_teacher;                  -- 关闭游标
    FOR i IN 1.. teacher_table.COUNT LOOP  -- 显示教师信息
        DBMS_OUTPUT.PUT_LINE ('教师姓名: '|| teacher_table(i). teaName
        || ' 所在院系: '|| teacher_table(i). dept);
    END LOOP;
END;
```

这段PL/SQL语句块是将院系编号为t\_10的计算机系中的所有教师的信息显示输出。为了在FETCH语句中将游标结果集中的所有行一次性地都提取出来，这里使用了BULK COLLECT子句。最后将查询到的教师信息全部显示出来。其显示结果如下：

```
教师姓名: 张昌 所在院系: 计算机系
教师姓名: 赵伟 所在院系: 计算机系
教师姓名: 毛翠 所在院系: 计算机系
教师姓名: 于波 所在院系: 计算机系
```

### 20.2.6 在游标中使用子查询

在游标中还可以使用子查询。所谓子查询，是指将一个SELECT查询语句块嵌套在另一个SQL查询语句中。下面来看一个在游标中使用子查询的例子。

```
DECLARE
    CURSOR cursor_teacher IS
        SELECT teaID,teaName,age,sex,dept,profession,salary
        FROM T_teacher
        WHERE salary >ANY
        (SELECT salary
         FROM T_teacher
         WHERE dept = '数学系')
        AND dept != '数学系'
        ORDER BY salary ASC
BEGIN
    -- 执行游标操作代码
END;
```

这段PL/SQL语句块中，在游标中使用子查询，这里的查询语句是查询其他院系的教师中工资比任意一个数学系教师的工资都高的教师信息。

## 20.3 游标属性

游标属性主要是用来确定有关数据操作的执行信息。在PL/SQL中，游标属性主要包括%FOUND、%NOTFOUND、%ISOPEN和%ROWCOUNT四个属性。在PL/SQL语句块中可以使用游标属性，但是游标属性在SQL语句中不能使用。本节就来介绍这几个游标属性的用法。最后还会对显式游标属性和隐式游标属性做一个比较。

### 20.3.1 显式游标属性

显式游标的属性主要包括%FOUND、%NOTFOUND、%ISOPEN和%ROWCOUNT四种。通过使用显式游标属性可以返回多行查询结果。下面分别来介绍显式游标的这四种属性。

#### 1. %FOUND属性

%FOUND属性用来判断在游标的结果集中是否有数据记录存在。如果在FETCH语句中取得了一行数据记录，则%FOUND就会返回TRUE；如果FETCH语句中提取数据失败，则%FOUND会返回FALSE；如果在游标已经打开但是还没有提取数据之前，则%FOUND会返回NULL。下面来看一个使用%FOUND属性的例子。

```
BEGIN
    OPEN cursor_teacher;           -- 打开游标
    LOOP
        FETCH cursor_teacher       -- 提取数据
        INTO v_teaID, v_teaName,
        IF cursor_teacher %FOUND THEN -- 如果游标结果集中存在数据
            DBMS_OUTPUT.PUT_LINE ('教师编号: ' || v_teaID || '教师姓名: ' || v_teaName );
        ELSE                       -- 如果游标结果集中没有数据
    END;
```

```
        EXIT;                                -- 退出循环
    END IF;
END LOOP;
CLOSE cursor_teacher;                        -- 关闭游标
END;
```

在这个PL/SQL的代码片段中，说明了%FOUND属性的用法。在BEGIN部分中，在LOOP循环里使用%FOUND属性对cursor\_teacher游标中的结果集进行判断，如果游标结果集中存在数据记录，则将该信息取出并显示；如果游标结果集中不存在数据记录，则退出LOOP循环。

**注意** 在游标尚未打开，或者游标已经关闭的情况下使用了%FOUND属性，则会产生INVALID\_CURSOR的预定义异常信息。

## 2. %NOTFOUND

%NOTFOUND属性的作用和%FOUND属性的作用正好相反。如果在FETCH语句中取得了一行数据记录，则%NOTFOUND就会返回FALSE；如果FETCH语句中提取数据失败，则%NOTFOUND会返回TRUE；如果在游标已经打开但是还没有提取数据之前，则%NOTFOUND会返回NULL。下面来看一个使用%NOTFOUND属性的例子。

```
BEGIN
    OPEN cursor_teacher;                    -- 打开游标
    LOOP
        FETCH cursor_teacher                -- 提取数据
            INTO v_teaID, v_teaName,
        EXIT WHEN cursor_teacher %NOTFOUND; -- 如果游标结果集中没有数据就退出循环
        DBMS_OUTPUT.PUT_LINE ('教师编号: ' || v_teaID || '教师姓名: ' || v_teaName );
    END LOOP;
    CLOSE cursor_teacher;                    -- 关闭游标
END;
```

在这个PL/SQL的代码片段中，说明了%NOTFOUND属性的用法。在BEGIN部分中，在LOOP循环里使用%NOTFOUND属性对cursor\_teacher游标中的结果集进行判断，如果cursor\_teacher %NOTFOUND返回的结果为TRUE，则说明FETCH语句中提取数据失败，此时就需要退出LOOP循环；否则，如果cursor\_teacher %NOTFOUND返回的结果为FALSE，则说明游标结果集中含有数据记录，此时就将游标结果集中的信息显示输出。

读者可能会发现，在上面的PL/SQL的代码片段中，EXIT WHEN cursor\_teacher %NOTFOUND语句放在FETCH语句之后，数据处理之前，这样就保证了PL/SQL在整个处理过程中不会处理NOTFOUND属性值为NULL的情况。

有时，为了保证PL/SQL程序运行的安全性，需要cursor\_teacher %NOTFOUND返回的结果为NULL的情况，这时可以对上面的判断语句做如下的改进。

```
EXIT WHEN cursor_teacher %NOTFOUND OR cursor_teacher %NOTFOUND IS NULL;
```

在这个LOOP循环EXIT WHEN的判断语句中，在原来的EXIT WHEN cursor\_teacher %NOTFOUND的基础上又添加了一个cursor\_teacher %NOTFOUND IS NULL的判断条件，并使用OR关键字将这两个判断条件联系在一起。这段改进后的判断语句表示，如果cursor\_teacher %NOTFOUND返回的结果为

TRUE或者为NULL，就退出LOOP循环。

**注意** 在游标尚未打开，或者游标已经关闭的情况下使用了%NOTFOUND属性，则会产生INVALID\_CURSOR的预定义异常信息。

### 3. %ISOPEN属性

%ISOPEN属性用来判断游标是否已经被打开。如果对应的游标被打开，则%ISOPEN返回TRUE；如果对应的游标没有被打开，则返回FALSE。下面来看一个使用%ISOPEN属性的例子。

```
BEGIN
  IF cursor_teacher %ISOPEN THEN          -- 如果游标已经打开
    ...                                   -- 执行相应的操作
  ELSE
    OPEN cursor_teacher;                  -- 打开游标
  END IF;
END;
```

在这个PL/SQL的代码片段中，说明了%ISOPEN属性的用法。在BEGIN部分中，IF-ELSE语句对cursor\_teacher游标是否打开做出判断。如果cursor\_teacher游标已经打开，就执行相应的操作；如果cursor\_teacher游标没有打开，就使用OPEN语句将cursor\_teacher游标打开。

除了可以使用上面的IF-ELSE语句对cursor\_teacher游标是否打开做出判断以外，还可以使用下面的形式判断cursor\_teacher游标是否打开。

```
BEGIN
  IF NOT cursor_teacher %ISOPEN THEN      -- 如果游标没有打开
    OPEN cursor_teacher;                  -- 打开游标
  END IF
  ...                                     -- 执行相应的操作
END;
```

### 4. %ROWCOUNT属性

%ROWCOUNT属性用来取得游标取得的实际数据记录的行数。它返回的是游标已经检索的数据记录的行数。在游标变量已经打开但是没有检索数组之前，%ROWCOUNT值为0；当FETCH语句取出了第一条数据记录时，%ROWCOUNT的值为1；当FETCH语句取出了第二条数据记录时，%ROWCOUNT的值就会加1，变为2，依次类推，每当从FETCH语句中成功取出一条数据记录后，%ROWCOUNT的值就加1。下面来看一个使用%ROWCOUNT属性的例子。

```
BEGIN
  OPEN cursor_teacher;                    -- 打开游标
  LOOP
    FETCH cursor_teacher                   -- 提取数据
      INTO v_teaID, v_teaName,
    EXIT WHEN cursor_teacher %NOTFOUND OR cursor_teacher %ROWCOUNT > 10;
    DBMS_OUTPUT.PUT_LINE ('教师编号: ' || v_teaID || '教师姓名: ' || v_teaName);
  END LOOP;
  CLOSE cursor_teacher;                   -- 关闭游标
END;
```

在这个PL/SQL的代码片段中，说明了% ROWCOUNT属性的用法。在BEGIN部分中，对cursor\_teacher游标进行判断，如果cursor\_teacher游标中没有取出数据记录或者取得的数据记录超过了10行，就退出LOOP循环。

**注意** 在游标尚未打开，或者游标已经关闭的情况下使用了%ROWCOUNT属性，则会产生INVALID\_CURSOR的预定义异常信息。

### 20.3.2 隐式游标属性

隐式游标是指在使用INSERT、UPDATE、DELETE和SELECT INTO语句执行相关更新和查询操作时的SQL游标。隐式游标属性会返回这些INSERT、UPDATE、DELETE和SELECT INTO语句的相关执行信息。在Oracle执行完相关更新或者查询语句之后，会自动关闭SQL游标。

同显式游标一样，隐式游标也有%FOUND、%ISOPEN、%NOTFOUND和%ROWCOUNT四种。它们的使用方法与显式游标的使用方法相同，但是在使用隐式游标属性时，有两点是需要注意的。

- ❑ 在Oracle数据库打开SQL游标之前，隐式游标中的所有属性都为NULL。
- ❑ 在Oracle执行完相关更新或者查询语句之后，会自动关闭SQL游标，因此SQL游标的%ISOPEN属性的值总是FALSE。

下面来看一个使用隐式游标属性的例子。这个例子曾经在17.1.2小节中提到过，但是当时可能对这个例子中的一些细节并不是很清楚，在了解完隐式游标属性之后，再来看一下这个例子。

```
DECLARE
    v_teaID VARCHAR2(15) := 't152303';
    v_teaName VARCHAR2(10) := '王杰';
    v_age NUMBER(2) := 45;
    v_dept VARCHAR2(20) := '计算机系';
    v_profession VARCHAR2(10) := '教授';
    v_salsry NUMBER(6,2) := 5000;
BEGIN
    UPDATE t_teacher
    SET profession = v_profession
    WHERE teaID = v_teaID;
    IF SQL%NOTFOUND THEN
        -- 判断教师是否存在
        INSERT INTO T_student
        VALUES(v_teaID, v_teaName, v_age, v_dept, v_profession, v_salsry);
    END IF;
END;
```

在这个PL/SQL的语句块中，需要对教师信息表进行更新操作，因此这里使用了隐式游标属性%NOTFOUND对教师信息进行判断，如果该名教师在教师信息表中不存在，就会执行INSERT INTO语句，将该名教师的信息插入到教师信息表中；如果该名教师在教师信息表中已经存在，就会执行UPDATE SET语句，将该名教师的职称进行修改。

从这个例子中也可以看到，隐式游标属性不需要在DECLARE部分来声明，系统可以自动对其进行声明。如果需要使用隐式游标属性，在BEGIN部分直接使用就可以了。



### 20.3.3 显式游标属性与隐式游标属性比较

在20.3.1节和20.3.2节中，分别介绍了显式游标属性和隐式游标属性的使用方法，通过这上面两个小节的讲解，读者应该对显式游标属性和隐式游标属性有了一个基本的了解。这一节就对显式游标属性与隐式游标属性的一些不同之处做一些比较。

- ❑ 显式游标属性需要显式地对其进行声明、打开、提取数据和关闭的操作；而隐式游标属性是由系统自动声明的，没有相应的打开（OPEN）、提取数据（FETCH）和关闭（CLOSE）的操作，在BEGIN部分可以直接使用。
- ❑ 显式游标属性中在属性的%的前面是一个游标的名字（例如，cursor\_teacher % NOTFOUND）；而隐式游标属性中在属性的%的前面是SQL（例如，SQL%NOTFOUND）。
- ❑ 显式游标属性中的%ISOPEN属性，在对应的游标被打开的时候返回TRUE；在对应的游标没有被打开的时候返回FALSE；而隐式游标属性中的%ISOPEN属性的返回值始终是FALSE。
- ❑ 从执行效率来看，显式游标的执行效率要比隐式游标的执行效率高。

## 20.4 游标循环

在前面的讲解中，已经了解到在使用FETCH语句提取数据记录时，每次只能从结果集中提取一条记录，取得该条记录之后，将游标下移，指向当前记录的下一条记录。在实际应用中，一般都需要对结果集中的所有数据记录进行遍历，并对每一条数据记录进行处理，这就需要在FETCH语句中通过使用循环语句来实现。这一节就来介绍有关游标循环的内容。另外还会介绍如何使用游标修改和删除游标结果集中的数据以及参数化游标的使用。

### 20.4.1 简单LOOP循环

在19.3节中，已经了解到，PL/SQL代码中最简单的循环语句是LOOP循环，这里就从简单的LOOP循环开始，看一下如何使用简单LOOP循环对游标进行处理。

在使用简单LOOP循环处理游标时，一般使用显式游标属性来控制循环的次数。通过使用LOOP循环语句可以使用标量变量、记录变量或者集合变量（例如，PL/SQL表、嵌套表、可变数组）来读取游标结果集中的数据。

#### 1. 使用标量变量读取游标结果集中的数据

首先来看使用标量变量读取游标结果集中数据的例子。下面的这个例子是用来查询指定学生编号的学生的课程成绩。

```
DECLARE
    CURSOR cursor_stuResult IS                -- 声明游标
        SELECT S.stuName, C.curName,R.result
        FROM t_result R,t_curriculum C,t_student SC
        WHERE R.curID=C.curID
        AND   R.stuID=S.stuID
        AND   R.stuID = &stuID;
    v_stuName   t_student . stuID %TYPE;      -- 学生姓名
    v_curName   t_ curriculum. curName %TYPE;  -- 课程名称
    v_result    t_ result. result %TYPE;       -- 课程成绩
```

## 零基础学SQL

```
BEGIN
  OPEN cursor_stuResult;                -- 打开游标
  LOOP
    FETCH cursor_stuResult              -- 提取数据
      INTO v_stuVame, v-curName, v_result
    IF cursor_stuResult %FOUND THEN      -- 如果游标结果集中存在数据
      DBMS_OUTPUT.PUT_LINE ('教师编号: ' || v_v_stuName || '课程名称: ' || v_curName || '
课程成绩' v_result);
    ELSE                                 -- 如果游标结果集中没有数据
      EXIT;                             -- 退出循环
    END IF;
  END LOOP;
  CLOSE cursor_stuResult;               -- 关闭游标
END;
```

这段PL/SQL语句块是用来查询指定学生编号的学生的课程成绩。在DECLARE部分声明了一个cursor\_stuResult游标，并定义了三个标量变量v\_stuName、v\_curName、v\_result，分别用来表示学生姓名、课程名称和课程成绩。在BEGIN部分，首先使用OPEN语句打开游标，然后使用一个LOOP循环，将游标中的数据记录一一提取出来，如果cursor\_stuResult %NOTFOUND为TRUE（游标结果集中没有数据记录）的情况下，就退出LOOP循环；如果cursor\_stuResult %NOTFOUND为FALSE（游标结果集中含有数据记录），就读取游标结果集中的数据，并将得到的数据记录显示出来。其显示结果如下所示。

```
输入stuID的值: s102203
学生姓名: 赵亮 课程名称: 计算机系统结构 课程成绩: 85
学生姓名: 赵亮 课程名称: 数据库基础 课程成绩: 75
学生姓名: 赵亮 课程名称: C语言 课程成绩: 90
学生姓名: 赵亮 课程名称: 高等数学 课程成绩: 60
```

**注意** 使用标量变量读取游标结果集中的数据时，其标量变量的数据类型和长度必须与数据表中对应的列的数据类型和长度相匹配。

### 2. 使用记录变量读取游标结果集中的数据

在PL/SQL中，也可以使用记录变量读取游标结果集中的数据。下面来看通过记录变量读取游标结果集中数据的例子。下面的这个例子是通过使用记录变量来查询指定学生编号的学生的课程成绩。

```
DECLARE
  CURSOR cursor_stuResult IS           -- 声明游标
    SELECT S.stuName, C.curName, R.result
    FROM t_result R, t_curriculum C, t_student SC
    WHERE R.curID=C.curID
    AND    R.stuID=S.stuID
    AND    R.stuID = &stuID;
  stuResult_record cursor_stuResult;   -- 定义记录
BEGIN
  OPEN cursor_stuResult;               -- 打开游标
  LOOP
    FETCH cursor_stuResult              -- 提取数据
```

```
        INTO stuResult_record;
    EXIT WHEN cursor_stuResult %NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('学生姓名: ' || stuResult_record.stuName
                          || ' 课程名称: ' || stuResult_record.curName
                          || ' 课程成绩: ' || stuResult_record.result);

    END LOOP;
    CLOSE cursor_stuResult;          -- 关闭游标
END ;
```

这个PL/SQL语句块使用记录变量来查询指定学生编号的学生的课程成绩。在DECLARE部分定义了一个记录变量stuResult\_record，在BEGIN部分使用这个记录变量查询指定学生编号的学生课程成绩。同样这里也使用cursor\_stuResult %NOTFOUND对LOOP循环进行判断，在cursor\_stuResult %NOTFOUND为TRUE（游标结果集中没有数据记录）的情况下，就退出LOOP循环，否则就将查询到的数据记录一一显示出来。

```
输入stuID的值: s281234
学生姓名: 王龙 课程名称: 计算机系统结构 课程成绩: 93
学生姓名: 王龙 课程名称: 数据库基础 课程成绩: 71
学生姓名: 王龙 课程名称: C语言 课程成绩: 95
学生姓名: 王龙 课程名称: 高等数学 课程成绩: 90
```

### 3. 使用集合变量读取游标结果集中的数据

在PL/SQL中，还可以使用集合变量读取游标结果集中的数据。下面来看通过集合变量读取游标结果集中数据的例子。下面的这个例子是通过使用PL/SQL表来查询指定学生编号的学生的课程成绩。

```
DECLARE
    CURSOR cursor_stuResult IS          -- 声明游标
        SELECT S.stuName, C.curName,R.result
        FROM t_result R,t_curriculum C,t_student SC
        WHERE R.curID=C.curID
        AND    R.stuID=S.stuID
        AND    R.stuID = &stuID;
    TYPE stuResult_table_type IS TABLE OF cursor_stuResult %ROWTYPE
    INDEX BY BINARY_INTEGER;
    stuResult_table stuResult_table_type;  -- 定义PL/SQL表
    v_i INT;                             -- 定义变量
BEGIN
    OPEN cursor_stuResult;              -- 打开游标
    LOOP
        v_i = cursor_stuResult %ROWTYPE+1;
        FETCH cursor_stuResult          -- 提取数据
            INTO stuResult_table(v_i);
        EXIT WHEN cursor_stuResult %NOTFOUND;
        DBMS_OUTPUT.PUT_LINE ('学生姓名: ' || stuResult_table(v_i).stuName
                              || ' 课程名称: ' || stuResult_table(v_i).curName
                              || ' 课程成绩: ' || stuResult_table(v_i).result);

    END LOOP;
    CLOSE cursor_stuResult;             -- 关闭游标
END ;
```



这个PL/SQL语句块使用PL/SQL表来查询指定学生编号的学生的课程成绩。在DECLARE部分定义了一个PL/SQL表stuResult\_table，在BEGIN部分使用PL/SQL表查询指定学生编号的学生课程成绩。同样这里也使用cursor\_stuResult %NOTFOUND对LOOP循环进行判断，当cursor\_stuResult %NOTFOUND为TRUE（游标结果集中没有数据记录）时，就退出LOOP循环，否则就将查询到的数据记录一一显示出来。

```
输入stuID的值: s281234
学生姓名: 王龙 课程名称: 计算机系统结构 课程成绩: 93
学生姓名: 王龙 课程名称: 数据库基础 课程成绩: 71
学生姓名: 王龙 课程名称: C语言 课程成绩: 95
学生姓名: 王龙 课程名称: 高等数学 课程成绩: 90
```

从查询结果可以看到，当输入的学生编号值为s281234时，其显示的学生成绩结果与使用记录变量查询的学生编号为s281234显示的学生成绩相同。

## 20.4.2 WHILE循环

在PL/SQL中，也可以使用WHILE-LOOP循环语句处理游标。下面来看一个使用WHILE-LOOP循环语句处理游标的例子。这里还以查询指定学生编号的学生的课程成绩为例。

```
DECLARE
    CURSOR cursor_stuResult IS          -- 声明游标
        SELECT S.stuName, C.curName,R.result
        FROM t_result R,t_curriculum C,t_student SC
        WHERE R.curID=C.curID
        AND   R.stuID=S.stuID
        AND   R.stuID = &stuID;

    v_stuName   t_student . stuID %TYPE;    -- 学生姓名
    v_curName   t_curriculum. curName %TYPE; -- 课程名称
    v_result    t_result. result %TYPE;      -- 课程成绩
BEGIN
    OPEN cursor_stuResult;                -- 打开游标
    FETCH cursor_stuResult                -- 提取数据
        INTO v_stuName, v_curName, v_result
    WHILE cursor_stuResult %FOUND LOOP
        DBMS_OUTPUT.PUT_LINE ('学生姓名: '|| v_stuName
                                || ' 课程名称: '|| v_curName
                                || ' 课程成绩: '|| v_result);
        FETCH cursor_stuResult            -- 提取数据
            INTO v_stuName, v_curName, v_result
        END LOOP;
    CLOSE cursor_stuResult;                -- 关闭游标
END ;
```

这段PL/SQL语句块是用来查询指定学生编号的学生的课程成绩。在DECLARE部分与简单LOOP循环中的例子相同。在BEGIN部分，需要使用两次FETCH语句，在使用OPEN语句打开游标之后，WHILE-LOOP循环语句之前使用一次，在WHILE循环数据记录处理完成之后使用一次。这是为了保证将每一个查询数据记录都要作为循环条件进行判断。在WHILE-LOOP循环中，使用cursor\_stuResult %FOUND对游标结果集中的数据进行判断，如果%FOUND属性为TRUE，则表示游标结果集中含有数

据记录，就读取游标结果集中的数据，并将得到的数据记录显示出来；如果%FOUND属性为FALSE，则表示游标结果集中没有数据记录，则跳出WHILE循环。

### 20.4.3 游标FOR循环

在前面的两种循环方式下使用游标都需要经历打开游标、提取数据和关闭游标这几个过程，为了简化游标的处理过程，可以使用游标FOR循环。它可以隐含地处理游标的打开、提取数据和游标的关闭操作。当开始进行循环时，游标会自动打开，在循环过程中，游标FOR循环会隐式地定义一个%ROWTYPE类型的记录，并把它作为FOR循环的循环索引，然后系统会自动读取游标结果集中当前的数据记录，将该数据记录放到对应的记录索引中，在循环处理结束之后，退出FOR循环时，系统就会自动地关闭游标。

**注意** 由于在使用游标FOR循环处理游标结果集中的数据时，可以隐含地处理游标的打开、提取数据和游标的关闭操作。因此在使用游标FOR循环时，就不能再使用OPEN语句、FETCH语句和CLOSE语句了。

下面以查询学生编号为s102203的学生的课程成绩为例，来看一个使用游标FOR循环是如何处理游标结果集中的数据的。

```
DECLARE
    CURSOR cursor_stuResult IS                -- 声明游标
        SELECT S.stuName, C.curName,R.result
        FROM t_result R,t_curriculum C,t_student SC
        WHERE R.curID=C.curID
        AND    R.stuID=S.stuID
        AND    R.stuID = 's102203';
BEGIN
    FOR stuResult_record IN cursor_stuResult LOOP
        DBMS_OUTPUT.PUT_LINE ('学生姓名: ' || stuResult_record .stuName
                                || ' 课程名称: ' || stuResult_record .curName
                                || ' 课程成绩: ' || stuResult_record .result);
    END LOOP;
END ;
```

在这段PL/SQL语句块中，使用游标FOR循环显示指定学生编号的学生的课程成绩。在DECLARE部分只是声明了一个cursor\_stuResult游标。在BEGIN部分，使用了一个游标FOR循环处理cursor\_stuResult中的结果集。其中，stuResult\_record是一个基于游标的隐含定义的%ROWTYPE类型的记录变量，该变量是编译器隐式定义的，该记录变量用来作为FOR循环的循环索引。这个记录变量的作用域范围只是在FOR循环内，在循环之外不能对其进行引用。

在FOR循环内，当循环开始之前，系统会自动打开cursor\_stuResult游标；在每一次循环时，会对满足条件的数据记录执行查询操作，并将该数据记录放到对应的记录索引中，当所有满足条件的数据记录都查询完成之后，会退出FOR循环，同时系统会自动地将cursor\_stuResult游标关闭。

将上面的这段PL/SQL语句块与使用LOOP循环和WHILE-LOOP循环的PL/SQL语句块相比可知，使用游标FOR循环只需要很少的语句就可以完成相应的功能，实现过程也变得简单了许多。

使用上面的游标FOR循环查询指定学生编号的学生的课程成绩，其PL/SQL的代码已经简化了很多。



但是，在游标FOR循环中如果不需要引用游标属性，那么游标FOR循环可以进一步地简化，即省略游标声明，在FOR循环中直接使用查询语句。例如对于前面讲到的查询指定学生编号的学生的课程成绩这个例子，如果在游标FOR循环中不引用游标属性，则可以使用下面的代码完成。

```
BEGIN
  FOR stuResult_record IN(SELECT S.stuName, C.curName,R.result
    FROM t_result R,t_curriculum C,t_student SC
    WHERE R.curID=C.curID
    AND   R.stuID=S.stuID
    AND   R.stuID = 's102203') LOOP
    DBMS_OUTPUT.PUT_LINE ('学生姓名: '|| stuResult_record .stuName
      || ' 课程名称: '|| stuResult_record .curName
      || ' 课程成绩: '|| stuResult_record .result);
  END LOOP;
END;
```

在这段PL/SQL语句块中，只包括BEGIN部分。在FOR循环中，将查询语句放到了FOR循环语句中关键字IN后面的括号内，并省略了游标声明。FOR循环中的stuResult\_record记录变量和游标都是隐式声明的。

在SQL语句中，讲过在SELECT语句中可以使用表达式。由于在游标FOR循环中，记录中域的名称和数据表中对应列的名称是相对应的（例如stuResult\_record.stuName与t\_student表中的列stuName相对应），因此如果在SELECT语句中使用了表达式，则需要为该列给定一个列别名。在FOR循环中如果希望引用该记录对应的域，就可以使用列别名来代替。例如，在下面的这个游标声明中，其查询语句里包含了一个表达式。

```
CURSOR cursor_stuResult IS                                -- 声明游标
  SELECT stuID,curID,(result+10) newresult
  FROM t_result R,
  WHERE R.stuID = 's102203';
```

这里声明了一个cursor\_stuResult游标，在其对应的SELECT查询语句中，查询的是成绩信息中学生编号为s102203的课程成绩，在这个查询语句中使用了一个表达式，将该学生的成绩加上10分，并为这个表达式起了一个别名newresult。此时，如果使用游标FOR循环将查询的结果显示出来，在BEGIN部分就可以使用如下的方法完成。

```
BEGIN
  FOR stuResult_record IN cursor_stuResult  LOOP
    DBMS_OUTPUT.PUT_LINE ('学生姓名: '|| stuResult_record .stuName
      || ' 课程编号: '|| stuResult_record . curID
      || ' 课程成绩: '|| stuResult_record . newresult);
  END LOOP;
END;
```

这里使用游标FOR循环将学生编号为s102203的课程成绩结果显示出来。可以看到，在显示课程成绩时，使用的是stuResult\_record.newresult，其中newresult就是上面查询语句中使用的列别名。

#### 20.4.4 使用游标修改和删除数据行

使用显式游标不仅是用来检索结果集中的数据记录，很多时候，在使用循环语句查询检索行的时

候，还希望对其进行相应的修改和删除操作。在PL/SQL中提供了这样的功能。要想使用游标修改和删除数据行，需要在游标声明时使用FOR UPDATE子句，在执行UPDATE或者DELETE语句时使用WHERE CURRENT OF子句。其语法规则如下：

```
CURSOR cursor_name IS select_statement;
    FOR UPDATE[OF column_reference][NOWAIT]
update_statement WHERE CURRENT OF cursor_name;
delete_statement WHERE CURRENT OF cursor_name
```

其中，cursor\_name表示指定游标的名字；select\_statement是一个与cursor\_name游标对应的查询语句；column\_reference表示被锁定的数据表中的列，被锁定的数据表的列可以是一列，也可以是多列，多列之间需要用逗号分开；关键字NOWAIT表示如果被请求的数据行已经被其他会话锁定，那么就无须再等待了；update\_statement表示用于修改的UPDATE语句；delete\_statement表示用于删除的DELETE语句。

很明显，使用游标修改和删除数据行的语法包括了两个部分，一部分是在游标声明时使用FOR UPDATE子句。如果声明的游标使用了FOR UPDATE子句，那么当使用OPEN语句打开游标时，就会把满足查询条件的用于修改和删除的数据行一一锁定在结果集中。例如，下面的游标声明是合法的。

```
DECLARE
    CURSOR cursor_student IS
        SELECT R.stuID,C.curID, C.curName,R.result
        FROM t_result R,T_curriculum C
        WHERE R. result>80
    FOR UPDATE;
```

此时，如果其他的用户在cursor\_student游标的结果集的数据行中已经设置了锁，那么Oracle需要等到那个用户将锁释放之后，才可以进行相应的操作。如果不希望这样的等待，就可以使用NOWAIT关键字。

```
DECLARE
    CURSOR cursor_student IS
        SELECT R.stuID,C.curID, C.curName,R.result
        FROM t_result R,T_curriculum C
        WHERE R. result>80
    FOR UPDATE NOWAIT;
```

这里在FOR UPDATE子句后面加上了一个NOWAIT关键字，当使用了NOWAIT关键字之后，如果cursor\_student游标的结果集的数据行已经被其他用户加锁，就会返回一个如下的错误信息。

```
ORA -54: resource busy and acquire with NOWAIT specified
```

在这种情况下Oracle就不会继续等待了，而是把控制权转交给PL/SQL程序，此时可以重新打开游标或者修改查询语句对其他没有加锁的数据行进行操作。

在对数据表进行查询时，如果希望对某一个表中的数据行加锁，可以在FOR UPDATE OF子句的后面指定特定表中的行。例如下面这个例子。

```
CURSOR cursor_stuResult IS                                -- 声明游标
    SELECT S.stuName, C.curName,R.result
    FROM t_result R,t_curriculum C,t_student SC
```

```
WHERE R.curID=C.curID
AND R.stuID=S.stuID
AND S.stuName = '赵亮'
FOR UPDATE OF R.result;
```

在这个游标声明中，使用FOR UPDATE OF子句对t\_result表中的字段result加锁，当FOR UPDATE OF子句引用到该表中的这个字段的时候，该表中的行才会被锁定。上面的例子中，就把行锁定在t\_result表中。

使用游标修改和删除数据行的语法中的另外一部分是在执行UPDATE或者DELETE语句时使用WHERE CURRENT OF子句。可以使用UPDATE或者DELETE语句的CURRENT OF子句引用从指定游标中取出的结果集中的数据记录。例如下面的例子。

```
DECLARE
    CURSOR cursor_teaSalsry IS                -- 声明游标
        SELECT teaID, teaName,salary
        FROM t_teacher FOR UPDATE;
BEGIN
    FOR teacher_record IN cursor_ teaSalsry LOOP
        DBMS_OUTPUT.PUT_LINE ('教师编号: ' || stuResult_record . teaID
                                || ' 教师姓名: ' || stuResult_record . teaName
                                || ' 教师工资: ' || stuResult_record . salary);
        UPDATE t_teacher SET salary = salary+200      -- 修改教师工资信息
        WHERE CURRENT OF cursor_ teaSalsry;
    END LOOP;
    COMMIT;                                          -- 提交
END ;
```

这段PL/SQL语句中是使用显式游标修改教师工资。在DECLARE部分声明游标时，使用了FOR UPDATE子句，表示在循环语句中要对检索的数据记录进行更新操作。在BEGIN部分，使用游标FOR循环将检索到的教师信息一一显示输出，同时使用UPDATE语句更新教师信息表中的教师工资信息，将每一位教师的工资在原有工资的基础上添加200元，WHERE CURRENT OF cursor\_ teaSalsry子句中的cursor\_teaSalsry是FOR UPDATE子句中指定的游标名。最后使用COMMIT语句将修改后的数据记录提交。提交（或者回滚）之后，对数据行的锁定也会解除。

**注意** 当声明一个被UPDATE或者DELETE语句的子句WHERE CURRENT OF中引用的游标时（例如，cursor\_teaSalsry），就必须使用FOR UPDATE子句指定游标。如果没有在FOR UPDATE子句中指定的游标上使用WHERE CURRENT OF子句，是不合法的。

#### 20.4.5 参数化游标

游标也可以接收参数。使用带参数的游标需要在定义游标时为其提供参数并要指定参数的数据类型。其语法规则如下：

```
CURSOR cursor_name(parameter_name dataType) IS select_statement;
```

其中，cursor\_name为游标的名字；parameter\_name 表示指定的参数；dataType用来指定该参数的数据类型。参数可以定义一个，也可以定义多个，多个参数之间需要用逗号分开。select\_statement是一

个与cursor\_name游标对应的查询语句。

```
DECLARE
    CURSOR cursor_teacher(v_deptID t_teacher. deptID%TYPE ,
                          v_ profession t_teacher. profession %TYPE)
    SELECT teaID,teaName,salary
    FROM t_teacher
    WHERE deptID = v_deptID
    AND profession= v_ profession;
```

在DECLARE部分的游标声明中，使用了两个参数。其中，v\_deptID用来表示教师所在的院系；v\_ profession用来表示教师的职称。

这里需要说明的一点是，游标参数的作用域是本地的，也就是说这些参数只能在游标声明时对应的SELECT语句中使用。如果希望使用游标的参数，需要使用OPEN语句。OPEN语句用来为游标传递参数，将实际的值传递给游标。在使用OPEN语句时，其声明的实际参数要与游标声明中的每一个形式参数相对应。也就是说，在OPEN语句中需要为游标声明中的每一个形式参数都指定一个对应的实际参数。使用OPEN语句传递参数的语法规则如下：

```
OPEN cursor_name(parameter_value);
```

其中，parameter\_value是与游标声明中的每一个形式参数相对应实际参数的值。例如，对于上面的cursor\_teacher的游标声明，使用OPEN语句打开cursor\_teacher的语句如下所示。

```
OPEN cursor_teacher ('t_10','教授');
```

这里是使用OPEN语句打开cursor\_teacher游标。其中，把值“t\_10”传递给参数v\_deptID，把值“教授”传递给参数v\_ profession。

上面介绍的是显式游标传递参数的方法。在游标循环中讲到游标FOR循环可以隐含地处理游标的打开、提取数据和游标的关闭操作。在游标FOR循环中不能再使用OPEN语句、FETCH语句和CLOSE语句。那么在游标FOR循环中是否可以把参数传递给游标呢？答案是肯定的。可以在游标FOR循环中把参数传递给游标。下面通过一个例子来看一下如何在游标FOR循环中将参数传递给游标。

```
DECLARE
    CURSOR cursor_teacher(v_deptID t_teacher. deptID%TYPE ,
                          v_ profession t_teacher. profession %TYPE)
    SELECT teaID,teaName,salary
    FROM t_teacher
    WHERE deptID = v_deptID
    AND profession= v_ profession;
BEGIN
    FOR teacher_record IN cursor_teacher('t_10','教授') LOOP
        DBMS_OUTPUT.PUT_LINE ('教师编号: '|| teacher_record . teaID
                                || ' 教师姓名: '|| teacher_record . teaName
                                || ' 教师工资: '|| teacher_record . salary);;
    END LOOP;
END;
```

这段PL/SQL语句块中是查询院系编号为t\_10（即计算机系）、职称为教授的教师信息。在DECLARE部分声明了一个cursor\_teacher游标，并使用了两个参数。其中，v\_deptID用来表示教师所在



## 零基础学SQL

的院系；v\_ profession用来表示教师的职称。在BEGIN部分，在游标FOR循环中向cursor\_teacher游标传递两个参数，并将两个实际的值传递给游标，其中，“t\_10”表示院系编号，“教授”表示教师的职称。其查询结果如下所示。

```
教师编号: t103265 教师姓名: 张昌 教师工资: 3800
教师编号: t106358 教师姓名: 毛翠 教师工资: 4000
```

## 20.5 使用游标变量

前面讲到的显式游标定义的都是静态游标。静态游标与某个SQL语句相关联，它是按照游标打开时SQL语句所查询的记录作为显示的结果集，对于游标打开以后数据库中所做的更新操作并不会显示出来。也就是说，对于INSERT、UPDATE和DELETE语句所做的更新操作，静态游标并不会将其显示在已经打开的结果集中。游标变量可以在打开游标时指定其对应的SELECT语句，这可以实现动态游标，对于使用INSERT、UPDATE和DELETE语句所做的更新操作在动态游标中都是可见的。这一节就来介绍使用游标变量的方法。最后还将介绍使用游标变量的一些限制。

### 20.5.1 声明游标变量

游标变量的类型是REF CURSOR，它是一种引用类型，类似于C语言中的指针。在程序运行时，可以指向不同的内存地址。在使用游标变量之前，需要首先声明一个游标变量。其声明游标变量的语法规则如下：

```
TYPE type_name IS REF CURSOR [RETURN return_type];
cursorVariable_name type_name;
```

其中，type\_name表示REF CURSOR的类型名；RETURN return\_type表示指定的REF CURSOR返回的选择列表的记录类型；cursorVariable\_name用来指定游标变量的名字。

**注意** 游标变量中，RETURN子句中的返回类型return\_type必须是记录类型。该记录类型可以显式声明，也可以使用%ROWTYPE隐式声明。

下面来看两个声明游标变量的例子。首先来看一个没有RETURN子句的游标变量的声明。如果在声明游标变量时没有RETURN子句，那么在打开该游标变量时，其查询的SELECT语句可以任意指定。

```
DECLARE
TYPE teaSalary_cursorRef IS REF CURSOR;
v_ teaSalary teaSalary_cursorRef;
```

这里声明的v\_ teaSalary游标变量是REF CURSOR类型的。其中，teaSalary\_cursorRef是这个REF CURSOR类型的名字。

上面的DECLARE部分声明的游标变量v\_teaSalary是一个没有RETURN子句的游标变量。当然也可以为其声明一个含有特定返回类型的游标变量。

```
DECLARE
TYPE teacher_cursorRef IS REF CURSOR RETURN t_teacher%ROWTYPE;
v_teacher teacher_cursorRef;
```

这里声明的v\_teacher游标变量是REF CURSOR类型的。这个游标声明中包含有一个RETURN子句。



其中，`t_teacher%ROWTYPE`指定了该游标变量返回的选择列表的记录类型；`teacher_cursorRef`是这个REF CURSOR类型的名字。

RETURN子句中的记录类型除了可以使用%ROWTYPE隐式声明之外，也可以通过自定义记录的方式显式地声明。

```
DECLARE
    TYPE t_studetRec IS RECORD(
        -- 定义记录
        v_stuName      t_student . stuName%TYPE ,    -- 学生姓名
        v_age          t_student . age%TYPE,         -- 学生年龄
        v_sex          t_student . sex%TYPE E,        -- 学生性别
    );
    student_record t_studetRec;
    TYPE student_cursorRef IS REF CURSOR RETURN student_record %TYPE;
    v_student student_cursorRef;
```

这里在DECLARE部分自定义了一个t\_studetRec记录，然后声明了一个游标变量v\_student，在声明游标变量时使用RETURN子句将自定义的t\_studetRec记录作为该游标变量的返回类型。

**注意** 在声明游标变量时，如果使用RETURN子句指定其返回类型，那么在打开该游标变量时，其查询的SELECT语句中的选择列表就必须与游标指定的返回类型相匹配。否则，系统就会出现ROWTYPE不匹配的ROWTYPE\_MISMATCH预定义异常信息。

### 20.5.2 打开游标变量

在声明游标变量之后，就可以打开一个游标变量。打开游标变量是使用OPEN语句来完成的，在使用OPEN语句打开游标变量时，需要指定游标变量对应的SELECT查询语句。其打开游标变量的语法规则如下：

```
OPEN cursorVariable_name FOR select_statement;
```

其中，`cursorVariable_name`表示已经声明的游标变量的名字；`select_statement`表示游标变量执行时对应的SELECT查询语句。

在20.5.1小节声明游标变量中已经提到过，如果在声明游标变量时使用RETURN子句指定其返回类型，那么在打开该游标变量时，其查询的SELECT语句中的选择列表就必须与游标指定的返回类型相匹配。所以如果在声明游标变量时使用RETURN子句指定其返回类型，那么使用OPEN语句打开游标时其对应的SELECT查询语句就不能任意地指定了，其SELECT查询语句的选择列表就必须与游标指定的返回类型相匹配。

例如，20.5.1小节中游标变量v\_teacher就是一个在声明时使用了RETURN子句指定其返回类型的游标变量，当打开该游标变量时，可以使用如下的方式。

```
OPEN v_teacher FOR
    SELECT* FROM t_teacher;
```

### 20.5.3 取得数据结果

在完成了声明游标和打开游标变量的操作之后，就可以通过游标变量检索结果集中的数据了。使用游标变量检索结果集中的数据是使用FETCH语句来完成的。其语法规则如下：

```
FETCH cursorVariable_name INTO variable1 [variable2...]
```

其中，cursorVariable\_name为前面已经打开的游标变量的名字；variable1、variable2表示的是声明的变量。这里的变量可以是一个标量变量，也可以是一个记录或者PL/SQL中的集合变量。多个变量之间需要用逗号分隔。

下面来看一个使用FETCH语句的例子。例如，如果要使用FETCH语句提取游标变量v\_teacher中的数据记录，可以使用下面的方法完成。

```
FETCH v_teacher INTO teacher_record;
```

这里的v\_teacher是在前面已经声明并打开的游标变量，teacher\_record表示的是一个记录类型的变量。该记录变量可以在DECLARE部分定义。

```
teacher_record t_teacher%ROWTYPE;
```

#### 20.5.4 关闭游标变量

当查询的结果集检索完成之后，就可以关闭游标变量了。关闭游标变量是使用CLOSE语句来完成的。关闭游标变量后，与游标变量相关的资源将会被释放。关闭游标变量的语法规则如下：

```
CLOSE cursorVariable _name;
```

其中，cursorVariable \_name表示关闭游标变量的名字。这里的cursor cursorVariable \_name是前面已经打开的游标变量名。游标变量被关闭后还可以重新将其打开。

#### 20.5.5 一个使用游标变量的例子

在前面的几个小节中，分别介绍了使用游标变量需要包括的包括4个过程：声明游标变量、打开游标变量、提取数据和关闭游标变量。这一节通过一个例子来看一下如何使用游标变量检索数据。这个例子中，是通过游标变量查询教师信息表中的教师信息，如果教师信息表中存在教师信息，就将这些教师信息插入到new\_teacher表中。

```
DECLARE
    TYPE teacher_cursorRef IS REF CURSOR RETURN t_teacher%ROWTYPE;
    v_teacher teacher_cursorRef;                -- 声明游标变量
    teacher_record t_teacher%ROWTYPE;           -- 定义记录
BEGIN
    OPEN v_teacher FOR                          -- 打开游标变量
        SELECT*
        FROM t_teacher;
    LOOP
        FETCH v_teacher INTO teacher_record;    -- 提取数据
        EXIT WHEN v_teacher%NOTFOUND;
        /*执行插入操作*/
        INSERT INTO new_teacher(teaID, teaName, age, sex, deptID, dept, profession, salary, pension)
        VALUES(teacher_record.teaID, teacher_record.teaName, teacher_record.age, teacher_record.sex,
            teacher_record.deptID, teacher_record.dept, teacher_record.profession,
            teacher_record.salary, teacher_record.pension);
    END LOOP;
    CLOSE v_teacher;                            -- 关闭游标变量
```

```
COMMIT;                                -- 提交数据
END;
```

这段PL/SQL语句块中，通过使用游标变量查询教师信息表中的教师信息，并将查询到的教师信息插入到new\_teacher表中。在DECLARE部分声明了一个名为v\_teacher的游标变量，该游标变量包含一个RETURN子句，指定其返回类型。在BEGIN部分，使用OPEN语句打开游标变量v\_teacher，在LOOP循环中从teacher\_record记录中提取数据，并将查询到的数据插入到new\_teacher表中。当v\_teacher%NOTFOUND为TRUE，即在游标变量的结果集中没有数据记录时，退出LOOP循环。最后使用CLOSE语句关闭游标变量，并使用COMMIT语句将更新的数据提交。

上面是一个简单的使用游标变量的方法。在使用游标变量时，还可以在同一个游标变量中返回不同类型的数据。例如下面的这个例子。

```
BEGIN
  IF table_name = 'student' THEN
    OPEN v_cursor FOR                                -- 打开游标
    SELECT v_stuID, v_stuName
    FROM t_student;
  ELSE
    OPEN v_cursor FOR                                -- 打开游标
    SELECT v_teaID, v_teaName
    FROM t_teacher;
  LOOP
    IF table_name = 'student' THEN
      FETCH v_cursor INTO v_stuID, v_stuName;        -- 提取数据
      EXIT WHEN v_cursor %NOTFOUND;
      DBMS_OUTPUT.PUT_LINE ('学生编号: ' || v_stuID || ' 学生姓名: ' || v_stuName);
    ELSE
      FETCH v_teacher INTO v_teaID, v_teaName;        -- 提取数据
      EXIT WHEN v_cursor %NOTFOUND;
      DBMS_OUTPUT.PUT_LINE ('教师编号: ' || v_teaID || ' 教师姓名: ' || v_teaName);
    END IF;
  END LOOP;
  CLOSE v_cursor;                                    -- 关闭游标
END;
```

在这个PL/SQL片段中，只有一个游标变量v\_cursor，在BEGIN部分，对table\_name进行判断，根据table\_name取得值的不同，让游标变量v\_cursor返回不同类型的数据，并在LOOP循环中将得到的数据记录显示输出。

游标变量还可以应用在Pro\*C程序以及存储过程中。另外，在向PL/SQL程序传递主游标变量时，如果把多个OPEN-FOR语句组合在一起使用，还可以减少网络流量。

### 20.5.6 使用游标变量的一些限制

从上面的例子中，可以看到使用游标变量可以简化PL/SQL的处理过程。但是，游标变量在使用时也会有一些限制。

❑ 游标变量不能在包中声明。例如，下面的声明是非法的。

```
CREATE OR REPLACE PACKAGE SEIJO AS
```

```
TYPE teacher_cursorRef IS REF CURSOR RETURN t_teacher%ROWTYPE;
v_teacher teacher_cursorRef; -- 包中不允许声明游标变量
```

其中，CREATE OR REPLACE PACKAGE SEIJO表示一个包。有关包的内容可以参看第24章。

- ❑ 游标变量不能被赋空值，游标变量是否相等或者是否为空也不能通过比较运算符来判断。
- ❑ 数据库中的表和视图、PL/SQL的集合（例如，PL/SQL表、嵌套表、可变数组）中不能存放游标变量值。
- ❑ 游标变量和游标之间不能相互替换。
- ❑ 远端子程序中不能使用游标变量。也就是说，游标变量不能从一个服务器传递到另一个服务器，但是游标变量可以在客户端和服务器的PL/SQL中进行传递。

## 20.6 嵌套游标

在Oracle9i及其以后的版本中，还可以使用嵌套游标。嵌套游标可以通过使用CURSOR表达式来实现，它可以用来处理PL/SQL语句块中一些复杂的查询操作。一个CURSOR表达式可以返回一个嵌套游标。声明一个CURSOR表达式的语法规则如下：

```
CURSOR ( subquery )
```

其中，subquery表示子查询语句。CURSOR表达式可以作为游标声明、REF CURSOR声明和游标变量的查询的一部分，可以通过嵌套循环的方式处理嵌套游标中的数据。嵌套游标会在父级游标取得数据时隐式地打开，除了可以显式地关闭嵌套游标外，在父级游标被关闭、取消、重新执行或者读取数据发生的情况下，嵌套游标也会自动被关闭。

下面来看一个使用嵌套游标的例子。在这个例子中是用来查找选择课程编号为t105对应的课程名称以及选择这门课程的学生编号，在从t\_curriculum表中取得课程编号对应的课程名称的同时，从另一张表t\_result中取得选择这门课的学生编号及其对应的成绩。

```
DECLARE
    TYPE refcursor_type IS REF CURSOR;
    CURSOR cursor_curName IS
        SELECT C.curName,
               CURSOR(SELECT *
                       FROM t_result
                       WHERE curID = C.curID
                      )
        FROM t_curriculum C
        WHERE C.curID = 't105';
    v_result refcursor_type; -- 声明游标变量
    stuResult_record t_result %ROWTYPE; -- 定义记录
    v_curName t_curriculum.curName %TYPE; -- 定义表示课程名称的变量
BEGIN
    OPEN cursor_curID; -- 打开游标
    LOOP
        FETCH cursor_curName -- 提取数据
        INTO v_result, cursor_curriculum;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.put_line ('课程名' || v_curName || ' '); -- 显示课程名
```

```
LOOP
    FETCH v_result
        INTO stuResult_record;                -- 提取数据
    EXIT WHEN loccur%NOTFOUND;
    DBMS_OUTPUT.put_line ( '选课学生ID ' || stuResult_record.stuID || ' '
                          '课程成绩' || stuResult_record.result);
END LOOP;
END LOOP;
CLOSE cursor_curName;                        -- 关闭游标
END;
```

在这个PL/SQL语句块中，在DECLARE部分，在声明cursor\_curName游标的同时，使用CURSOR表达式定义了一个嵌套游标，并声明了一个v\_result游标变量，并定义了一个stuResult\_record记录和一个表示课程名称的标量变量v\_curName。在BEGIN部分，使用OPEN语句打开cursor\_curID游标，cursor\_curName游标的结果集中，每一行都包含有一条记录，在这一条记录中，每一个列都对应包含有一个值，在这些列包含的值中其中有一部分会包含嵌套游标。因此，这里需要通过嵌套的LOOP循环取得嵌套游标中的数据。

从这个例子中，可以看出PL/SQL语句块中对嵌套游标的处理方式。通过使用嵌套循环首先处理游标结果集中的数据，然后再处理每一行中的嵌套游标。

在使用CURSOR表达式时，同样会有一些限制。例如，CURSOR表达式只能出现在查询语句的SELECT语句的列表中；CURSOR表达式不能在隐式游标中使用；在视图声明中不能使用CURSOR表达式等。

## 20.7 小结

本章主要介绍了游标的使用。对于显式游标和游标变量来说，都需要经过声明游标、打开游标、提取数据和关闭游标这4个过程。其中，OPEN语句用来打开游标；FETCH语句用来提取游标中的数据；CLOSE语句用于关闭游标。如果希望批量地提取数据可以使用BULK COLLECT子句。

显式游标和隐式游标属性主要包括%FOUND、%NOTFOUND、%ISOPEN和%ROWCOUNT四个属性，通过使用游标属性还可以确定游标当前的状态。

使用FETCH语句提取数据记录时，每次只能从结果集中提取一条记录，如果需要对结果集中每一条数据记录进行处理，需要在FETCH语句中通过使用循环语句来实现。本章还介绍了使用游标循环的方法，包括使用简单的LOOP循环、WHILE循环以及游标FOR循环的使用。

通过在游标声明时使用FOR UPDATE子句，在执行UPDATE或者DELETE语句时使用WHERE CURRENT OF子句可以对数据行执行修改和删除的操作。另外，游标也可以接收参数，还可以使用CURSOR表达式实现嵌套游标。



## 第21章 异常处理

在PL/SQL程序中，一般会遇到两种错误：一种是编译错误，这种错误是由PL/SQL引擎来检测；另外一种是运行时错误，这种错误需要通过PL/SQL中的异常处理机制进行捕获和处理。PL/SQL中的异常包括预定义异常和用户自定义的异常。预定义异常与一些常见的Oracle错误相关联。例如，为一个未初始化（NULL）的对象属性赋值、除数的值为0等；用户自定义的异常是程序开发人员根据应用程序的需要自定义的错误。本章就来介绍PL/SQL中有关异常处理的相关内容。

本章重点：

- ☐ 使用异常处理的原因
- ☐ 如何声明异常
- ☐ 如何抛出异常
- ☐ 如何捕获和处理异常
- ☐ 异常的处理机制
- ☐ 使用异常的原则

### 21.1 为什么要使用异常处理

在以前的PL/SQL程序中，对于程序运行时可能出现的问题都是通过使用IF-THEN条件语句进行判断，例如下面的这个例子。

```
DECLARE
    v_curName VARCHAR2(15);           -- 课程名字
    v_credit NUMBER(1);               -- 课程学分所在院系
    v_teacherName VARCHAR2(10);       -- 教师姓名
    v_teaID VARCHAR2(15);             -- 教师编号
    v_dept VARCHAR2(15);              -- 教师所在院系
    v_profession VARCHAR2(10);        -- 教师职称
    v_x NUMBER(2) :=30;               -- 教师年龄下限
    v_y NUMBER(2) :=50;               -- 教师年龄上限
BEGIN
    /*查询用户输入的课程名对应的课程信息*/
    SELECT curName,credit INTO v_curName, v_credit,
    FROM t_curriculum
    WHERE curName = & curName,
    IF SQL%NOTFOUND THEN              -- 判断课程信息是否存在
        DBMS_OUTPUT.PUT_LINE ('查询的信息不存在');
    END IF;
    /*查询年龄在30到50岁之间的教师信息*/
    SELECT teaID,teaName,dept,profession INTO v_teaID,v_teacherName,v_dept,v_profession
```

```
FROM t_teacher
WHERE age BETWEEN v_x AND v_y;
IF SQL%NOTFOUND THEN          -- 判断教师是否存在
    DBMS_OUTPUT.PUT_LINE ('查询的信息不存在');
END IF;
END;
```

在这段PL/SQL语句块中，在BEGIN部分有两个SELECT查询语句，分别用来查询用户输入的课程名对应的课程信息和年龄在30到50岁之间的教师信息，在这两个查询语句中分别要对SELECT查询语句的查询内容进行判断，如果查询信息为空，需要输出“查询的信息不存在”错误信息。

从上面的PL/SQL语句块中可以看到，在使用IF-THEN条件语句判断程序运行过程中可能出现的异常时，会带来下面两个问题。

- 对于复杂的PL/SQL应用程序来说，可能会出现很多种情况，如果使用IF-THEN条件语句对其每一个可能出错的条件进行判断，会使代码显得冗长，如果考虑不周还可能会遗漏某些判断条件，可能会引起其他的一些无关的错误。
- 将实现操作的代码和错误处理的代码放到一起使用，程序的逻辑结构不够清晰，也不利于代码的阅读。

为了解决这样的问题，就需要使用PL/SQL的异常处理。PL/SQL的异常处理方式与高级语言（例如，Java）的异常处理类似，在PL/SQL的程序中也可以抛出、捕获异常，并可以对捕获的异常进行处理。

在PL/SQL的程序中如果使用异常，就可以很方便地对程序可能产生的错误进行控制，也不需要编写检查错误的代码，这些问题都交给PL/SQL中的异常处理来完成。例如，对于上面的PL/SQL语句块，如果使用了PL/SQL中的异常处理，就可以使用下面的方法完成。

```
DECLARE
    v_curName VARCHAR2(15);          -- 课程名字
    v_credit NUMBER(1);              -- 课程学分所在院系
    v_teacherName VARCHAR2(10);      -- 教师姓名
    v_teaID VARCHAR2(15);            -- 教师编号
    v_dept VARCHAR2(15);             -- 教师所在院系
    v_profession VARCHAR2(10);       -- 教师职称
    v_x NUMBER(2) :=30;              -- 教师年龄下限
    v_y NUMBER(2) :=50;              -- 教师年龄上限
BEGIN
    /*查询用户输入的课程名对应的课程信息*/
    SELECT curName,credit , teacherName INTO v_curName, v_credit,
    FROM t_curriculum
    WHERE curName == & curName;
    /*查询年龄在30到50岁之间的教师信息*/
    SELECT teaID,teaName,dept,profession INTO v_teaID,v_teacherName,v_dept,v_profession
    FROM T_teacher
    WHERE age BETWEEN v_x AND v_y;
    EXCEPTION                      -- 异常处理部分
    WHEN NO_DATA_FOUND THEN        -- 判断查询信息是否存在
        DBMS_OUTPUT.PUT_LINE ('查询的信息不存在');
    END ;
END;
```

这段PL/SQL语句块是使用异常处理的程序，可以看到，在两个SELECT的查询语句中，已经不需要再使用IF-THEN条件语句对查询信息是否存在进行判断了，把这些判断的内容都交由EXCEPTION部分来处理，EXCEPTION部分中执行的是异常处理的语句。

可以看到，PL/SQL程序中对抛出异常的捕获和处理是通过在EXCEPTION部分编写异常控制代码来完成的。在EXCEPTION异常处理部分对SELECT查询语句的查询信息进行判断，如果有异常被抛出，也可以保证该异常能够被捕获并得到相应的处理。

比较这两段PL/SQL语句块可以发现，使用了PL/SQL中的异常处理之后，不需要再为程序中每一个可能出现的错误进行判断，编写错误判断代码，只需要在PL/SQL中的EXCEPTION部分编写异常控制代码就可以了。另外，异常处理部分将程序的执行代码与错误控制分离开来，使得程序的逻辑结构更加清晰，也利于代码的阅读和维护。

## 21.2 声明异常

PL/SQL中，异常声明只能在PL/SQL语句块、子程序或者包的声明部分。PL/SQL中的异常包括用户自定义异常和预定义异常两种类型。用户自定义异常需要在PL/SQL语句块的声明部分进行声明。下面是一个声明用户自定义异常的例子。

```
DECLARE
    e_illegalValue EXCEPTION;
```

其中，e\_illegalValue是一个标识符，EXCEPTION关键字表示该标识符是一个用户自定义的异常。一般声明一个异常都是以字母e开头的。

与用户自定义异常不同，预定义异常则不需要在声明部分声明，在Oracle数据库中也有很多的预定义异常，每一个预定义的异常都有对应的Oracle错误（有关预定义异常及其对应的错误可以参看21.4.2小节）。

异常的作用域与变量的作用域规则是一致的，即在声明部分中声明的用户自定义异常与在声明部分中定义的其他变量的作用域是相同的。这里需要说明的一点是，在嵌套的PL/SQL语句块中，如果在内层的PL/SQL语句块中声明的用户自定义异常（这里称为本地声明的异常）与外层PL/SQL语句块中声明的用户自定义异常（这里称为全局声明的异常）有相同的名字，那么在内层PL/SQL语句块中引用的就是本地声明的异常，而不会使用外层PL/SQL语句块中的那个全局声明的异常。例如，下面这个例子。

```
DECLARE
    e_illegalValue EXCEPTION;           -- 全局声明的异常
BEGIN
    DECLARE
        e_illegalValue EXCEPTION;      -- 内层的PL/SQL语句块
        -- 本地声明的异常
    BEGIN
        ...
        IF ... THEN
            RAISE e_illegalValue;       -- 本地声明的异常
        END IF;
    END;
EXCEPTION
    WHEN e_illegalValue THEN           -- 全局声明的异常
```

END;

这里在内层的PL/SQL语句块中使用RAISE语句抛出一个用户自定义的e\_illegalValue异常，在外层的PL/SQL语句块中使用WHEN e\_illegalValue THEN处理全局声明的异常。这里在内层PL/SQL语句块和在外层PL/SQL语句块中处理的是两个不同的e\_illegalValue异常，虽然这两个异常有相同的名字。（有关异常处理的内容可以参看21.5节）

## 21.3 抛出异常

在PL/SQL中，当PL/SQL语句块有错误发生时，运行时系统将异常抛出，就不能再回到当前语句块的可执行部分（BEGIN），而是将程序的控制权交给PL/SQL语句块或者是子程序的异常处理部分（EXCEPTION）进行处理。其中，用户自定义的异常必须使用RAISE语句显式地抛出；而通过编译指示EXCEPTION\_INIT与Oracle某个错误编号关联起来的用户自定义异常可以由运行时系统隐式地抛出；预定义异常既可以使用RAISE语句显式地抛出，也可以由运行时系统隐式地抛出。

这里首先来看一下使用RAISE语句显式抛出用户自定义异常的例子。在这个例子中，使用RAISE语句抛出一个用户自定义的异常e\_illegalValue。

```
DECLARE
    e_illegalValue EXCEPTION;
    v_result NUMBER (3);
BEGIN
    /*查询学生编号为s102203，课程编号为t105的成绩*/
    SELECT result INTO v_result
    FROM t_result
    WHERE stuID = 's102203'
    AND curID = t105';
    /*当查询到的学生成绩大于100 或者小于0时，抛出异常*/
    IF v_result >100 OR v_result <0 THEN
        RAISE e_illegalValue;
    END IF;
END;
```

在这个PL/SQL语句块中，在DECLARE部分首先声明一个用户自定义异常e\_illegalValue并定义了一个用于表示课程成绩的标量变量；在BEGIN部分使用SELECT语句查询学生编号为s102203，课程编号为t105的成绩，并对查询成绩进行判断，如果成绩大于100或者小于0，就使用RAISE语句显式抛出e\_illegalValue异常。如果希望对抛出的异常进行处理，可以在EXCEPTION部分使用WHEN语句（有关自定义异常处理的内容可以参看21.4.3）。

预定义异常也可以使用RAISE语句显式地抛出，下面来看一个使用RAISE语句显式抛出预定义异常的例子。

```
DECLARE
    v_resultPoin NUMBER (3);          -- 成绩绩点
    v_credit NUMBER (1);              -- 课程学分
BEGIN
    /*查询学生编号为s102203，课程编号为t105的成绩绩点*/
```

```
SELECT R.result / C.credit , C.credit INTO v_ resultPoin.,v_credit
FROM t_result R ,t_curriculum C
WHERE R curID. =C. curID
AND stuID = 's102203'
AND curID = t105';
/*当查询到的课程学分=0时，抛出异常*/
IF v_credit = 0 THEN
    RAISE ZERO_DIVIDE;
END IF;
END;
```

在这个PL/SQL语句块中，在DECLARE部分定义了两个标量变量，分别用来表示成绩绩点和课程学分；在BEGIN部分使用SELECT语句查询学生编号为s102203，课程编号为t105的课程成绩和该课程对应的学分，并对查询到的课程学分的值进行判断，如果课程学分的值为0，就使用RAISE语句显式抛出ZERO\_DIVIDE异常。ZERO\_DIVIDE异常是一个预定义异常，如果除数为0，则会抛出该异常；如果希望对抛出的异常进行处理，可以在EXCEPTION部分使用WHEN语句（有关异常处理的内容可以参看21.4.2）。

当然，预定义异常也可以不使用RAISE语句显式地抛出，可以由运行时系统隐式地抛出。例如，对于上面的例子，如果不使用RAISE语句显式地抛出，也可以使用下面的方式完成。

```
DECLARE
    v_ resultPoin    NUMBER (3);           -- 成绩绩点
    v_credit    NUMBER (1);               -- 课程学分
BEGIN
    /*查询学生编号为s102203，课程编号为t105的成绩绩点*/
    SELECT R.result / C.credit , C.credit INTO v_ resultPoin.,v_credit
    FROM t_result R ,t_curriculum C
    WHERE R curID. =C. curID
    AND stuID = 's102203'
    AND curID = t105';
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE ('课程学分值不能为0');
END;
```

由于ZERO\_DIVIDE异常是一个预定义异常，所以可以直接在EXCEPTION部分中使用WHEN语句对该异常进行捕获和处理（有关异常处理的内容可以参看21.5节）。

## 21.4 捕获和处理异常

PL/SQL中异常主要包括两种类型：预定义异常和用户自定义异常。在上一节中介绍了如何在PL/SQL语句块中抛出异常，同其他高级语言一样，如果在程序中抛出了异常，就需要对其进行捕获和处理。这一节就来介绍如何捕获和处理PL/SQL语句块中抛出的异常。

### 21.4.1 捕获和处理异常的语法规则

在PL/SQL中，异常的捕获和处理是由异常处理（EXCEPTION）部分完成的。异常处理的语法规



则如下：

```
EXCEPTION
    WHEN exception_name THEN
        statements1;
    WHEN exception_name THEN
        statements2;
    [WHEN OTHERS THEN
        statements3;]
```

其中，EXCEPTION关键字表示PL/SQL中异常处理部分的语句；WHEN子句后面的exception\_name表示异常的名字，这个异常的名字可以是预定义异常，也可以是用户自定义的异常；statements1、statements2表示的是当该异常产生时要执行的语句。最后的WHEN OTHERS THEN子句是可选的，OTHERS是PL/SQL中定义的一个特殊的异常处理器，处理那些WHEN子句没有处理的异常，类似于Java语言中的Exception类。由于WHEN OTHERS THEN子句将捕获所有的异常，包括预定义异常和用户自定义的异常，所以应该把它放到异常处理语句块中的最后，作为最后一个WHEN子句，保证其他的异常可以优先被捕获和处理。

**注意** 为了保证PL/SQL中所有的错误都会被检测到，应该在异常处理部分的最后使用WHEN OTHERS THEN子句捕获所有的异常。

了解了PL/SQL中的异常捕获和处理的方法之后，就可以在程序中对这些异常进行捕获和处理了。在下面的几个小节中将介绍如何在EXCEPTION部分处理预定义异常和自定义异常以及使用内置函数处理异常的方法。

21.4.2 处理预定义异常

预定义异常就是指Oracle数据库中提供的一些常见的Oracle错误，每一个系统异常都与一个特定的Oracle错误相对应，每一个Oracle错误都有一个错误编号。对于PL/SQL中的预定义异常，需要通过异常名称进行捕获和处理。PL/SQL在STANDARD包中定义了这些预定义异常，所以在使用这些预定义异常时，不需要再在声明部分（DECLARE）对其进行声明，而是可以直接使用。在表21.1中列出了主要的预定义异常和对应的错误编号及对这些预定义异常的简要说明。

表21.1 主要的预定义异常和对应的错误编号及其简要说明

预定义异常	错误编号	简要说明
ACCESS_INTO_NULL	ORA-06530	为一个未初始化(NULL)的对象属性赋值
CASE_NOT_FOUND	ORA-06592	CASE语句中没有满足条件的WHEN子句并且无ELSE子句
COLLECTION_IS_NULL	ORA-06531	在赋值之前没有对嵌套表或变长数组进行初始化操作，或者调用了未初始化的嵌套表或变长数组的集合方法
CURSOR_ALREADY_OPEN	ORA-06511	打开一个已经打开的游标
DUP_VAL_ON_INDEX	ORA-00001	向数据库中有唯一约束条件对应的列中插入重复的值
INVALID_CURSOR	ORA-01001	游标操作不合法
INVALID_NUMBER	ORA-01722	SQL语句中，字符转换数字失败
LOGIN_DENIED	ORA-01017	登录Oracle时使用了无效用户名、密码



(续)

预定义异常	错误编号	简要说明
NO_DATA_FOUND	ORA-01403	SELECT INTO语句中没有返回数据；对嵌套表中被删除的元素或是PL/SQL表中未初始化的元素进行引用
NOT_LOGGED_ON	ORA-01012	没有与Oracle数据库建立连接
PROGRAM_ERROR	ORA-06501	PL/SQL程序内部发生错误
ROWTYPE_MISMATCH	ORA-06504	主游标变量和PL/SQL游标变量的数据行的数据类型不匹配
SELF_IS_NULL	ORA-30625	在空对象实例上调用其成员方法
STORAGE_ERROR	ORA-06500	PL/SQL运行时内存溢出或者内存空间不足
SUBSCRIPT_BEYOND_COUNT	ORA-06533	对嵌套表或变长数组元素引用时，其使用的下标索引值超过了嵌套表或变长数组中元素的数量范围
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	对嵌套表或变长数组元素引用时，其使用的下标索引值不合法（例如-1）
SYS_INVALID_ROWID	ORA-01410	使用无效的字符串向ROWID转换失败
TIMEOUT_ON_RESOURCE	ORA-00051	Oracle等待资源出现超时的情况
TOO_MANY_ROWS	ORA-01422	SELECT INTO语句返回数据行超过一行
VALUE_ERROR	ORA-06502	算术运算、类型转换、截位操作或对长度约束出现错误
ZERO_DIVIDE	ORA-01476	除数的值为0

表21.1中对列出的预定义异常都进行了一个简要的说明，下面对其中的几个预定义异常再做一下补充说明。

- ❑ INVALID\_CURSOR：当游标的操作不合法时，会出现该异常。例如，对一个未打开的游标执行关闭操作；在一个尚未打开的游标中执行数据的提取操作；关闭一个已经关闭的游标等。
- ❑ INVALID\_NUMBER：当字符转换数字失败时，PL/SQL中内嵌的SQL语句会抛出该异常。由于在一个SQL语句中，一个字符或是一个字符串并不能表示一个有效的数字，因此在向数字转换的过程中就会发生INVALID\_NUMBER异常。例如，在教师信息表（t\_teacher）中，表示教师工资的列salary就是一个数字类型的值，如果要在INSERT INTO语句中向该列插入一个字符就会抛出INVALID\_NUMBER异常。在过程化语句中，如果出现类似的转换失败，则会抛出异常VALUE\_ERROR异常。
- ❑ NO\_DATA\_FOUND：当SELECT INTO语句中没有返回数据或者对嵌套表中被删除的元素或是PL/SQL表中未初始化的元素进行引用时，会出现该异常。但是在以下的情况下则不会抛出NO\_DATA\_FOUND异常。一种情况是在使用FETCH语句对游标中的结果集进行提取时，最后一次执行FETCH语句提取数据会得不到值，也不会抛出NO\_DATA\_FOUND异常；另一种情况是在SQL中使用了聚合函数（例如，MAX、MIN、AVG、SUM、COUNT等）。因为SQL中的聚合函数返回的值要么只有一个，要么取不到值，所以如果SELECT INTO语句中调用了聚合函数，也不会抛出NO\_DATA\_FOUND异常。例如下面这个例子。

```
BEGIN
...
SELECT MAX(salary) INTO v_salary
FROM t_teacher
WHERE teaI= 't103265';
```

```
IF SQL%NOTFOUND THEN
...
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN      -- 不会被执行
...
END;
```

- ❑ ROWTYPE\_MISMATCH：主游标变量和PL/SQL游标变量的数据行的数据类型不匹配时，会发生该异常。例如，将一个已经打开的主游标变量传递到一个存储过程中时，实参返回的数据类型和形参的数据类型出现了不匹配的现象，就会抛出该异常。
- ❑ VALUE\_ERROR：当在过程化语句中出现算术运算、类型转换、截位操作或者对长度约束等错误时，会出现该异常。例如，字符串转换成数字失败；在执行赋值操作时，数值的字符长度大于变量定义时的长度等都会抛出该异常。

了解了PL/SQL中的预定义异常之后，就可以在程序中对这些异常进行捕获和处理了。下面来看一个处理预定义异常的例子。在这个例子中，查询指定教师编号的教师工资，并对可能产生的错误进行捕获和处理。

```
DECLARE
    v_salary NUMBER(6,2);          -- 教师工资
BEGIN
    /* 查询指定教师编号的教师工资*/
    SELECT salary INTO v_salary
    FROM t_teacher
    WHERE teaID= ' t156354';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询数据不存在');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('teaID does not exist!',SYSDATE, 'admin');
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE ('查询的结果多于1行');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('teaID row is over 1!',SYSDATE, 'admin');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!',SYSDATE, 'admin');
END;
```

这段PL/SQL语句块中，在DECLARE部分定义了一个表示教师工资的标量变量；在BEGIN部分使用SELECT语句查询教师工资；在EXCEPTION部分对程序可能产生的异常进行了捕获和处理。这里需要捕获两个预定义的异常，一个是NO\_DATA\_FOUND，另一个是TOO\_MANY\_ROWS。在EXCEPTION部分的最后使用OTHERS来捕获在WHEN子句中没有捕获的异常，并对其进行处理。在使用WHEN子句捕获到异常之后，会显示相应的异常信息，并将该异常信息插入到t\_log表中。

上面的PL/SQL语句块中，每一个WHEN子句都只有一个异常名。在PL/SQL中，一个WHEN子句中也可以包括多个异常名。例如，上面的PL/SQL语句块中，异常处理部分的语句也可以使用如下的方式

完成。

```
EXCEPTION
  WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE ('查询数据时出现错误');
    INSERT INTO t_log          -- 将错误信息写入日志表中
    VALUES('a select error!',SYSDATE, 'admin');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('发生其他错误');
    INSERT INTO t_log          -- 将错误信息写入日志表中
    VALUES('Oracle error!',SYSDATE, 'admin');
```

这里将两个异常NO\_DATA\_FOUND 和TOO\_MANY\_ROWS放在了一个WHEN子句中。在一个WHEN子句中可以处理多个异常，多个异常名之间可以使用OR关键字分开。在WHEN子句中处理预定义异常时需要注意以下几个问题。

- ☐ 异常名称只能在异常处理部分出现一次，它只能在EXCEPTION部分的其中一个WHEN子句进行处理。如果一个异常在多个WHEN子句中使用，则会出现编译错误。
- ☐ 在一个WHEN子句中可以处理多个异常，多个异常名之间可以使用OR关键字分开。但是关键字OTHERS只能单独使用。
- ☐ 异常处理的程序中可以引用本地变量或者全局变量。
- ☐ 在使用游标FOR循环时，如果游标FOR循环中抛出了异常，那么在异常处理程序调用之前，游标就会被隐式地关闭。

### 21.4.3 处理自定义异常

PL/SQL中可以使用用户自定义的异常。但是使用用户自定义异常与使用预定义异常不同，用户自定义异常必须显式地声明，而且需要使用RAISE语句显式地将其抛出。下面通过一个例子来看一下在PL/SQL中如何处理用户自定义的异常。

```
DECLARE
  e_ illegalValue  EXCEPTION;
  e_ noRows  EXCEPTION;
  v_ result    NUMBER (3);
BEGIN
  /*查询学生编号为s102203、课程编号为t105的成绩*/
  SELECT result INTO v_ result
  FROM t_result
  WHERE stuID ='s102203'
  AND curID =' t105';
  /*没有查询到数据，抛出e_ noRows异常*/
  IF SQL%NOTFOUND THEN
    RAISE e_ noRows;
  END IF;
  /*当查询到的学生成绩大于100 或者小于0时，抛出e_illegalValue异常*/
  IF v_ result >100 OR v_ result <0 THEN
    RAISE e_illegalValue;
  END IF;
EXCEPTION
```

```
WHEN e_noRows THEN
    DBMS_OUTPUT.PUT_LINE ('查询数据不存在');
    INSERT INTO t_log
        VALUES('result does not exist!',SYSDATE, 'admin');
WHEN e_illegalValue THEN
    DBMS_OUTPUT.PUT_LINE ('查询成绩不合法 (大于100或者小于0)');
    INSERT INTO t_log
        VALUES('result is illegal!',SYSDATE, 'admin');
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('发生其他错误');
    INSERT INTO t_log
        VALUES('Oracle error!',SYSDATE, 'admin');
END;
```

在这个PL/SQL语句块中，在DECLARE部分首先声明两个用户自定义异常e\_illegalValue和e\_noRows，并定义了一个用于表示课程成绩的标量变量；在BEGIN部分使用SELECT语句查询学生编号为s102203，课程编号为t105的成绩，并对查询成绩进行判断，如果没有查询到数据记录，就使用RAISE语句显式抛出e\_noRows异常；如果成绩大于100或者小于0，就使用RAISE语句显式抛出e\_illegalValue异常；在EXCEPTION部分对e\_illegalValue和e\_noRows异常分别进行捕获和处理，并将错误信息写入t\_log表中，最后使用OTHERS来捕获在WHEN子句中没有捕获的异常，并对其进行处理。

在PL/SQL中也可以在一个WHEN子句中处理多个用户自定义异常，多个异常名之间可以使用OR关键字分开。例如，上面的PL/SQL语句块中，异常处理部分的语句也可以使用如下的方式完成。

```
EXCEPTION
WHEN e_noRows OR e_illegalValue THEN
    DBMS_OUTPUT.PUT_LINE ('查询数据时出现错误');
    INSERT INTO t_log
        VALUES('a select error!',SYSDATE, 'admin');
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('发生其他错误');
    INSERT INTO t_log
        VALUES('Oracle error!',SYSDATE, 'admin');
```

这里将两个异常e\_noRows和e\_illegalValue放在了一个WHEN子句中，两个异常名之间使用OR关键字将其分开。在WHEN子句中处理自定义异常时需要注意以下几个问题。

- ☐ 异常名称只能在异常处理部分出现一次，它只能在EXCEPTION部分的其中一个WHEN子句进行处理。如果一个异常在多个WHEN子句中使用，则会出现编译错误。
- ☐ 在一个WHEN子句中可以处理多个异常，多个异常名之间可以使用OR关键字分开。但是关键字OTHERS只能单独使用。
- ☐ 异常处理的程序中可以引用本地变量或者全局变量。
- ☐ 在使用游标FOR循环时，如果游标FOR循环中抛出了异常，那么在异常处理程序调用之前，游标就会被隐式地关闭。

#### 21.4.4 使用内置函数处理异常

前面的例子中讲过使用WHEN OTHERS THEN子句可以捕获所有的错误，但是WHEN OTHERS



THEN子句只是捕获错误，并不记录到底产生的是哪个错误。有些时候，知道产生的是什么错误是很重要的，通过对错误编号和错误信息的捕获可以对应用程序进行进一步的分析，提高应用程序的健壮性。这里介绍几个可以记录Oracle错误编号和错误信息的内置函数。

### 1. SQLCODE函数和SQLERRM函数

在PL/SQL中，提供了这样的内置函数可以对捕获的Oracle错误编号和错误信息进行记录。其中，SQLCODE函数返回的是当前Oracle错误编号，SQLERRM函数返回的是当前错误信息。下面通过一个例子来看一下SQLCODE函数和SQLERRM函数的使用方法。

```
BEGIN
    /*插入学生信息*/
    INSERT INTO t_student
        ('s102203','赵亮',23,'男','19860516');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('错误编号:'||SQLCODE);
        DBMS_OUTPUT.PUT_LINE ('错误信息:'||SQLERRM);
END;
```

在这段PL/SQL语句块中，在BEGIN部分插入一条学生记录（这条学生记录在t\_student），在EXCEPTION部分，使用WHEN OTHERS THEN捕获异常，并使用SQLCODE函数和SQLERRM函数将其捕获到的错误编号和错误信息显示输出。其显示结果如下：

```
错误编号: -0001
错误信息: unique constraint (SCOTT.STUID_PK) violated
```

其中，-0001是错误编号，该错误编号对应的错误信息为“unique constraint (SCOTT.STUID\_PK) violated”，即插入的数据违反了主键的唯一性约束条件。从表21.1中可以查到，错误编号为-0001对应的错误名称为DUP\_VAL\_ON\_INDEX，当向数据库中有唯一约束条件对应的列中插入重复的值时，就会抛出该异常。

了解了SQLCODE函数和SQLERRM函数的使用方法，那么在PL/SQL进行异常处理时，就可以在WHEN OTHERS THEN子句后使用SQLCODE函数和SQLERRM函数记录产生错误的编号及其对应的信息了。

对于用户自定义的异常，也可以使用SQLCODE函数和SQLERRM函数。其中，SQLCODE函数返回的值为1；SQLERRM函数返回值是“‘admin’-defined Exception”。

### 2. RAISE\_APPLICATION\_ERROR函数

使用SQLCODE函数和SQLERRM函数可以返回Oracle的错误编号和对应的错误信息，而使用RAISE\_APPLICATION\_ERROR可以用来创建用户自定义的错误信息。RAISE\_APPLICATION\_ERROR函数是包DBMS\_STANDARD的一部分，对它的引用可以不添加限定修饰词，调用RAISE\_APPLICATION\_ERROR的语法规则如下：

```
RAISE_APPLICATION_ERROR (error_number, error_message[, {TRUE | FALSE}]);
```

其中，error\_number表示错误的编号，它是一个负整数，其取值范围为-20000~-20999；error\_message用来指定与该错误相关的错误信息的字符串，其最大长度为2048字节；第三个参数是一

个布尔值，它是可选的，如果第三个可选参数的值为TRUE，则表示将新的错误放到前面已存在错误的栈顶；如果第三个可选参数的值为FALSE，则表示用新的错误取代前面已存在的所有的错误信息。该参数的默认值为FALSE。

RAISE\_APPLICATION\_ERROR函数只能在一个正在执行的存储子程序或方法中被调用，RAISE\_APPLICATION\_ERROR函数被调用后，系统会将用户自定义的错误编号及其对应的错误信息返回给应用程序，同其他的Oracle错误一样，用户自定义的错误编号及其对应的错误信息也可以被系统捕获。下面来看一个使用RAISE\_APPLICATION\_ERROR函数的例子。

```
CREATE OR REPLACE PROCEDURE update_salary
(p_teaID t_teacher.teaID%TYPE,
p_rate NUMBER(4,2),
)
AS
    v_salary NUMBER(6,2);          -- 教师工资
BEGIN
    /* 查询指定教师编号的教师工资*/
    SELECT salary INTO v_salary
    FROM t_teacher
    WHERE teaID= p_teaID;
    /*如果教师工资为NULL*/
    IF v_salary IS NULL THEN
        RAISE RAISE_APPLICATION_ERROR(-20111,'salary is illegal');
    ELSE
        /*如果查询的教师工资存在*/
        UPDATE t_teacher
        SET salary = salary +salary * p_rate
        WHERE teaID = p_teaID;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE RAISE_APPLICATION_ERROR(-20112,' teaID does not exist!');
END update_salary;
```

这里的CREATE OR REPLACE PROCEDURE表示创建的是一个存储过程。（有关存储过程的详细内容可以参看第22章）在BEGIN部分，使用SELECT语句查询指定教师编号的教师工资，其中p\_teaID是存储过程update\_salary中的一个参数，可以通过调用方法向存储过程中传入该参数。如果查询到的教师工资为NULL，就使用RAISE语句抛出一个异常，该异常是一个使用RAISE\_APPLICATION\_ERROR函数的用户自定义异常。在RAISE\_APPLICATION\_ERROR函数里定义了错误编号及其对应的错误信息；如果查询到的教师工资存在，就将该教师工资进行修改，其中p\_rate也是存储过程update\_salary中的一个参数，可以通过调用方法向存储过程中传入该参数。

在EXCEPTION部分使用WHEN子句捕获NO\_DATA\_FOUND异常，并在捕获该异常后抛出一个一个使用RAISE\_APPLICATION\_ERROR函数的用户自定义异常。

#### 21.4.5 编译提示EXCEPTION\_INIT

在PL/SQL中，通过编译指示EXCEPTION\_INIT可以将一个异常名称和某个Oracle错误编号关联起

## 零基础学SQL

来，这样就可以通过异常名称来引用某个Oracle错误。与在关键字OTHERS中使用SQLCODE函数和SQLERRM函数不同，使用编译提示EXCEPTION\_INIT可以不通过关键字OTHERS进行捕获。在PL/SQL中使用编译提示EXCEPTION\_INIT的语法规则如下：

```
PRAGMA EXCEPTION_INIT(exception_name,- Oracle_error_number);
```

其中，exception\_name表示一个已经声明过的异常名称，该异常的声明要在编译提示声明之前进行；-Oracle\_error\_number用来指定一个与该异常相关联的Oracle错误编号。例如下面这个例子。

```
DECLARE
    e_illegalValue EXCEPTION;
    PRAGMA EXCEPTION_INIT (e_illegalValue, -6502);
BEGIN
    /*查询学生编号为s102203, 课程成绩*/
    SELECT result INTO v_result
    FROM t_result
    WHERE stuID = 's102203'
EXCEPTION
    WHEN e_illegalValue THEN
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('result is illegal!',SYSDATE, 'admin');
END;
```

在这个PL/SQL语句块中，在DECLARE部分首先声明一个用户自定义异常e\_illegalValue，然后使用编译提示EXCEPTION\_INIT将该异常与Oracle错误编号-6502相关联。如果在程序运行时出现ORA-6502的错误，则会发生e\_illegalValue用户自定义的异常。

## 21.5 异常处理机制

在PL/SQL语句块中，声明部分（DECLARE）、可执行部分（BEGIN）和异常处理部分（EXCEPTION）都可能会产生异常。那么在PL/SQL语句块的这几个部分中异常是如何进行处理的呢？这一节就来介绍在声明部分、可执行部分和异常处理部分中异常的处理机制。

### 21.5.1 声明部分中异常的处理机制

在一个嵌套的PL/SQL语句块中，如果内层的声明部分在赋值时产生了异常，那么该异常会被外层的EXCEPTION部分的异常处理语句所捕获。声明部分中异常的处理机制如图21.1所示。

从图21.1中可以看到，在内层的DECLARE部分中出现的错误不会在该语句块的异常处理部分进行处理，而是传递到外层语句块中的异常处理部分，由外层语句块的异常处理部分进行捕获和处理。例如下面的这个例子。

```
BEGIN
    DECLARE
        v_credit NUMBER(2) = 'ab';
    -- 出现异常
```

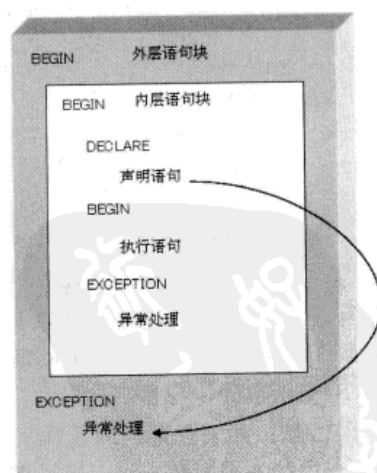


图21.1 声明部分中异常的处理机制

```
v_ curName VARCHAR2(10);
BEGIN
    SELECT curName INTO v_ curName
    FROM t_curriculum
    WHERE credit = v_ credit
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!',SYSDATE, 'admin');
END;
EXCEPTION
    WHEN OTHERS THEN          -- 捕获内层DECLARE部分出现的异常
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!',SYSDATE, 'admin');
END;
```

在这个PL/SQL语句块中，在内层语句块中DECLARE部分定义了一个表示学分的标量变量v\_ credit，并为其赋值为‘ab’。由于v\_ credit是一个NUMBER类型的变量，因此为其赋值为‘ab’会引发一个VALUE\_ERROR的预定义异常。该异常将会被外层的WHEN OTHERS THEN子句捕获，由外层的EXCEPTION部分来处理，而不会在内层语句块的EXCEPTION部分进行处理，即使内层语句块的EXCEPTION部分有WHEN OTHERS THEN子句。

### 21.5.2 可执行部分中异常的处理机制

在PL/SQL语句块的可执行部分如果出现异常，则首先会在当前的语句块中对异常进行处理，如果当前的语句块中没有找到与该异常相对应的处理程序，则该异常会被传递到外层语句块中，如果在外层语句块中有与该异常相对应的处理程序，则在外层语句块中对该异常进行捕获和处理，如果在外层语句块中也没有找到与该异常相对应的处理程序，依次类推，如果最外层语句块中还没有与该异常相对应的处理程序，那么PL/SQL会向主程序抛出一个未捕获异常。可执行部分中异常的处理机制如图21.2所示。

从图21.2中可以看到，在内层语句块中的异常处理部分只会处理内层语句块中执行部分所抛出的错误，例如这里抛出的错误A，会在内层语句块的异常处理部分进行捕获和处理。如果在内层语句块的异常处理部分没有捕获到该错误，例如错误B，PL/SQL就会将其传递到外层语句块的异常处理部分对其进行捕获和处理。对于在外层语句块也没有捕获到的异常，PL/SQL会向主程序抛出一个未捕获异常。下面通过一个例子来看一下在可执行部分中PL/SQL的异常处理机制。

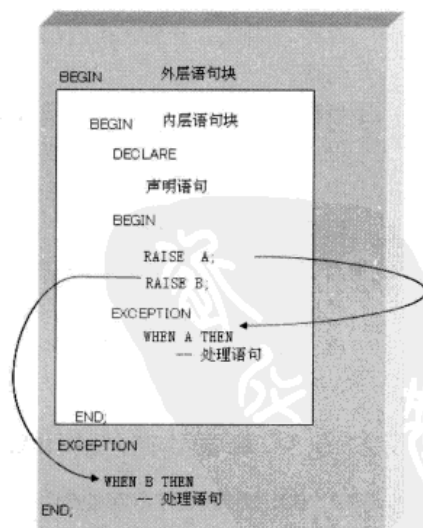


图21.2 可执行部分中异常的处理机制

DECLARE

```
A EXCEPTION;
B EXCEPTION;
C EXCEPTION;
v_number NUMBER(2);
BEGIN
    BEGIN
        IF v_number = 0 THEN
            RAISE A;                -- 抛出异常A
        ELSEIF v_number < 0 THEN
            RAISE B;                - 抛出异常B
        ELSE
            RAISE C;                - 抛出异常C
        END IF;
    EXCEPTION
        WHEN A THEN                -- 捕获异常A
            DBMS_OUTPUT.PUT_LINE ('the error is A');
    END;
EXCEPTION
    WHEN B THEN                    -- 捕获异常B
        DBMS_OUTPUT.PUT_LINE ('the error is B');
END;
```

这个PL/SQL片段在内层BEGIN部分在不同情况下可能会抛出异常A、异常B、异常C。在内层的EXCEPTION部分捕获异常A，在外层的EXCEPTION部分捕获了异常B。通过这个PL/SQL片段可以说在可执行部分中PL/SQL的异常处理机制。

- ❑ 当在内层BEGIN部分A异常被抛出时，由于在内层EXCEPTION部分有与该异常相对应的处理程序，因此，异常A在内层语句块中就会被处理，然后PL/SQL会将程序的控制权转移给外层语句块。
- ❑ 当在内层BEGIN部分B异常被抛出时，由于在内层EXCEPTION部分没有与该异常相对应的处理程序，因此，异常B就会被传递到外层语句块中，在外层语句块中寻找与异常B相对应的处理程序，在外层EXCEPTION部分对异常B进行捕获和处理，然后PL/SQL会将程序的控制权转移给主程序处理。
- ❑ 当在内层BEGIN部分C异常被抛出时，由于在内层EXCEPTION部分没有与该异常相对应的处理程序，因此，异常C就会被传递到外层语句块中，在外层语句块中寻找与异常C相对应的处理程序，在外层EXCEPTION部分也没有找到与异常C相对应的处理程序，则PL/SQL会将该异常抛给主程序进行捕获。

### 21.5.3 异常处理部分中异常的处理机制

在PL/SQL语句块的异常处理部分也可以产生异常。通过使用RAISE子句可以显式地抛出一个异常。当在内层的异常处理部分抛出异常时，该异常会被传递到外层语句块中进行处理。异常处理部分中异常的处理机制如图21.3所示。

从图21.3中可以看到，当在内层语句块中的异常处理部分抛出一个错误时，例如错误B，该错误不会被内层的异常处理部分捕获，而是传递到外层语句块中，由外层的异常处理部分进行捕获和处理。下面通过一个例子来看一下在异常处理部分中PL/SQL的异常处理机制。



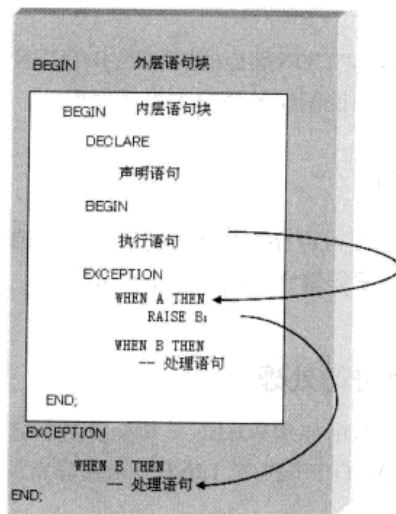


图21.3 异常处理部分中异常的处理机制

```
BEGIN
  DECLARE
    A EXCEPTION;
    B EXCEPTION;
    v_number NUMBER(2);
  BEGIN
    IF v_number = 0 THEN
      RAISE A;
    END IF;
  EXCEPTION
    WHEN A THEN
      RAISE B;
    WHEN B THEN
      DBMS_OUTPUT.PUT_LINE ('the error is B');
  END;
EXCEPTION
  WHEN B THEN
    DBMS_OUTPUT.PUT_LINE ('the error is B');
END;
```

这个PL/SQL片段在内层BEGIN部分在不同情况下可能会抛出异常A、异常B。在内层的EXCEPTION部分捕获异常A和异常B，在外层的EXCEPTION部分也捕获异常B。通过这个PL/SQL片段可以说明在异常处理部分中PL/SQL的异常处理机制。

- ❑ 当在内层BEGIN部分A异常被抛出时，由于在内层EXCEPTION部分有与该异常相对应的处理程序，因此，异常A在内层语句块中就会被处理。在内层EXCEPTION部分处理异常A时会抛出一个异常B。
- ❑ 当在内层EXCEPTION部分处理异常A时会抛出一个异常B，该异常B会被外层语句块的EXCEPTION部分的WHEN B THEN子句捕获，而不会被内层EXCEPTION部分的WHEN B

THEN子句捕获。

- 当异常B被外层语句块的EXCEPTION部分的WHEN B THEN子句捕获后，会对该异常进行处理，处理完成后PL/SQL会将程序的控制权转移给主程序处理。

## 21.6 使用异常处理原则

PL/SQL中通过使用异常处理可以对程序中出现的错误进行捕获和处理，但是在使用异常处理时也不能过于随便。使用异常处理时也需要遵守一些基本的使用原则。这一节将介绍有关使用异常处理的一些基本原则。

### 21.6.1 对捕获的异常要进行处理

在PL/SQL中如果在EXCEPTION中使用WHEN子句捕获到特定的异常，就需要编写处理语句对该异常进行处理。可以使用DBMS\_OUTPUT.PUT\_LINE显示该异常的相关信息，也可以将捕获到的异常信息插入到一个指定的数据表。

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('查询数据不存在');
    INSERT INTO t_log(          -- 将错误信息写入日志表中
      VALUES('teaID does not exist!',SYSDATE, 'admin'));
END;
```

在这个PL/SQL片段中，使用WHEN子句捕获NO\_DATA\_FOUND异常，在捕获到该异常之后，就将对应的异常信息显示输出，并使用INSERT INTO语句将该异常信息插入到t\_log数据表中。如果捕获了异常但是却处理该异常，那捕获它还有什么意义。

### 21.6.2 确保没有未处理的异常

一个PL/SQL程序中可能会产生多个异常，为了保证程序的健壮性，应该确保一个PL/SQL中所有的异常都会被捕获和处理。为了确保没有未处理的异常，可以在异常处理部分的最后使用WHEN OTHERS THEN子句捕获所有的异常。

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('查询数据不存在');
    INSERT INTO t_log          -- 将错误信息写入日志表中
      VALUES('teaID does not exist!',SYSDATE, 'admin');
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE ('查询的结果多于1行');
    INSERT INTO t_log          -- 将错误信息写入日志表中
      VALUES('teaID row is over 1!',SYSDATE, 'admin');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('发生其他错误');
    INSERT INTO t_log          -- 将错误信息写入日志表中
      VALUES('Oracle error!',SYSDATE, 'admin');
END;
```

在这个PL/SQL片段中，使用WHEN子句分别捕获NO\_DATA\_FOUND和TOO\_MANY\_ROWS异常，并在最后使用WHEN OTHERS THEN处理未捕获的其他异常。

### 21.6.3 标记异常发生的位置

在一个PL/SQL语句块中，可能会有多个SELECT查询语句，而每一个SELECT查询都需要捕获一个NO\_DATA\_FOUND异常，一旦程序产生了NO\_DATA\_FOUND异常，很难确定到底是哪一个SELECT查询语句产生的错误。例如下面的这个例子。

```
DECLARE
    v_curName VARCHAR2(15);           -- 课程名字
    v_credit NUMBER(1);               -- 课程学分所在院系
    v_teacherName VARCHAR2(10);       -- 教师姓名
    v_teaID VARCHAR2(15);             -- 教师编号
BEGIN
    /*查询用户输入的课程名对应的课程信息*/
    SELECT curName,credit , teacherName INTO v_curName, v_credit,
    FROM t_curriculum
    WHERE curName = '操作系统';
    /*查询年龄在30到50岁之间的教师信息*/
    SELECT teaID,teaName,dept,profession INTO v_teaID
    FROM t_teacher
    WHERE age BETWEEN 30 AND 50;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的信息不存在');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
END;
```

在这个PL/SQL的语句块中，BEGIN部分有两个SQL语句，分别用来查询用户输入的课程名对应的课程信息和年龄在30到50岁之间的教师信息；EXCEPTION部分捕获NO\_DATA\_FOUND异常，并显示相应的异常处理信息。但是，如果该程序中产生了NO\_DATA\_FOUND异常，则很难一下子确定到底是第一个SELECT查询语句产生的错误还是第二个SELECT查询语句产生的错误。

为了解决这个问题，可以在每一个SELECT语句中定义一个用来标记异常发生位置的变量，跟踪SELECT语句。

```
DECLARE
    v_curName VARCHAR2(15);           -- 课程名字
    v_credit NUMBER(1);               -- 课程学分所在院系
    v_teacherName VARCHAR2(10);       -- 教师姓名
    v_teaID VARCHAR2(15);             -- 教师编号
    v_counter INTEGER := 1;           -- 定义表示标记异常发生位置的变量
BEGIN
    /*查询用户输入的课程名对应的课程信息*/
    SELECT curName,credit INTO v_curName
    FROM t_curriculum
    WHERE curName = '操作系统';
    v_counter:= 2;                    -- 标记第一个SELECT语句
```



```
/*查询年龄在30到50岁之间的教师信息*/
SELECT teaID,teaName INTO v_teaID,v_teacherName
FROM t_teacher
WHERE age BETWEEN 30 AND 50,
v_counter:= 3; -- 标记第二个SELECT语句
EXCEPTION
WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('查询的信息不存在:' || v_counter);
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('发生其他错误');
END;
```

通过定义一个表示标记异常发生位置的变量，就可以很快地定位到产生NO\_DATA\_FOUND异常的SELECT语句。

除了可以在每一个SELECT语句中定义一个用来标记异常发生位置的变量，跟踪SELECT语句的方法之外，还可以将每一个SELECT查询语句都写在一个内层的子语句块中。在各自的语句块中分别捕获和处理异常。

```
DECLARE
    v_curName VARCHAR2(15); -- 课程名字
    v_credit NUMBER(1); -- 课程学分所在院系
    v_teacherName VARCHAR2(10); -- 教师姓名
    v_teaID VARCHAR2(15); -- 教师编号
BEGIN
    BEGIN
        /*查询用户输入的课程名对应的课程信息*/
        SELECT curName,credit INTO v_curName
        FROM t_curriculum
        WHERE curName = '操作系统';
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE ('查询的课程信息不存在');
            WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        END;
    BEGIN
        /*查询年龄在30到50岁之间的教师信息*/
        SELECT teaID,teaName,dept,profession INTO v_teaID,v_teacherName
        FROM t_teacher
        WHERE age BETWEEN 30 AND 50,
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE ('查询的教师信息不存在');
            WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        END;
    END;
END;
```



#### 21.6.4 控制异常处理程序

在PL/SQL中，当语句块中某一个语句出现异常时，就会进入到异常处理部分对异常进行捕获和处理，在异常处理完成之后，程序会继续向下执行而不会再重新回到当前的语句块。下面通过一个例子说明这个问题。

```
DECLARE
    v_resultPoin    NUMBER (3);           -- 成绩绩点
    v_credit         NUMBER (1);          -- 课程学分
BEGIN
    /*查询学生编号为s102203，课程编号为t105的成绩绩点*/
    SELECT R.result / C.credit , C.credit INTO v_resultPoin.,v_credit
    FROM t_result R ,t_curriculum C
    WHERE R curID. =C. curID
    AND stuID ='s102203'
    AND curID = t105';
    /*将成绩绩点插入到t_resultPoin 表中*/
    INSERT INTO t_resultPoin (resultPoin, stuID, curID)
    VALUES (v_resultPoin, 's102203', t105');
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE ('除数不能为0');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('the value of credit is 0!',SYSDATE, 'admin');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!',SYSDATE, 'admin');
END;
```

这段PL/SQL语句块中，在BEGIN部分首先查询学生编号为s102203，课程编号为t105的成绩绩点，然后将取得的该名学生的成绩绩点信息插入到一个数据表t\_resultPoin中；在EXCEPTION部分捕获并处理异常。由于在SELECT INTO语句涉及到了除法计算，所以可能会引起ZERO\_DIVIDE异常，因此这里需要对ZERO\_DIVIDE进行捕获和处理，最后使用WHEN OTHERS THEN子句处理其他未能捕获的异常。

这段PL/SQL语句块在执行时可能会出现这样一个问题，就是当SELECT INTO语句产生ZERO\_DIVIDE异常之后，程序会进行到EXCEPTION部分对该异常进行捕获和处理，处理完成之后，程序会正常结束，在SELECT INTO语句之后的INSERT INTO不能得到执行。

有些时候，根据程序应用的需要，希望在SELECT INTO语句产生ZERO\_DIVIDE异常之后，还可以执行之后的INSERT INTO语句的插入操作，应该如何处理呢？

如果希望在某一个语句产生异常之后，还可以执行其后面的操作，可以把可能产生异常的语句块单独放在内层的子块中，把该语句块后面要执行的操作代码放到该内层子块的外面，这样当该子块中的语句产生异常之后，在子块内就可以对异常进行捕获和处理，内层子块处理完异常之后，就会结束当前子块的操作，转而执行下一条执行语句。

对于上面的例子，如果希望在SELECT INTO语句产生异常之后，还能执行后面的INSERT语句的操作，就可以使用下面的代码来实现。



```
DECLARE
    v_resultPoin    NUMBER (3);          -- 成绩绩点
    v_credit        NUMBER (1);          -- 课程学分
BEGIN
    BEGIN
        /*查询学生编号为s102203, 课程编号为t105的成绩绩点*/
        SELECT R.result / C.credit , C.credit INTO v_resultPoin,v_credit
        FROM t_result R ,t_curriculum C
        WHERE R.curID. =C. curID
        AND stuID ='s102203'
        AND curID = t105';
    EXCEPTION
        WHEN ZERO_DIVIDE THEN
            v_resultPoin :=0;              -- 将成绩绩点设置为0
            DBMS_OUTPUT.PUT_LINE ('除数不能为0');
            INSERT INTO t_log              -- 将错误信息写入日志表中
            VALUES('the value of credit is 0!' ,SYSDATE, 'admin');
    END;
    /*将成绩绩点插入到t_resultPoin 表中*/
    INSERT INTO t_resultPoin (resultPoin, stuID, curID)
    VALUES (v_resultPoin, 's102203', t105');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log                -- 将错误信息写入日志表中
        VALUES('Oracle error!' ,SYSDATE, 'admin');
END;
```

在这段PL/SQL语句块中，把查询学生编号为s102203，课程编号为t105的成绩绩点的SELECT INTO语句放到了内层的BEGIN语句块中，在内层的子块中对ZERO\_DIVIDE异常进行捕获和处理。在外层的BEGIN语句块中执行INSERT INTO插入学生成绩绩点的代码，并在外层EXCEPTION部分使用WHEN OTHERS THEN子句对未捕获的异常进行处理。

经过上面的改写之后，可以看到，当内层子块中的SELECT INTO语句产生ZERO\_DIVIDE异常之后，在内层子块的EXCEPTION部分会对该异常进行捕获和处理，并将v\_resultPoin的值设置为0。该异常处理完成之后，会将程序的控制权转交给外层BEGIN语句块，继续执行INSERT INTO的插入操作。

## 21.7 小结

在实际开发应用程序的过程中，异常处理都是必不可少的一个重要环节。通过使用异常处理处理PL/SQL程序中可能出现的不同的异常情况，对于一个程序的健壮性也起着非常重要的作用。本章主要介绍了如何在PL/SQL中处理异常。包括声明异常、抛出异常、捕获和处理异常的方法，如何处理预定义异常和自定义异常以及PL/SQL中的异常处理机制，最后介绍了异常处理的一些原则。

## 第22章 存储过程

子程序指能够接受参数并可被其他程序调用来执行特定操作的PL/SQL块。PL/SQL中，子程序包括两种类型：过程和函数。这一章将介绍PL/SQL中子程序的存储过程。

存储过程可以认为是经过编译的，永久保存在数据中的一组SQL语句。通过创建和使用存储过程，可以提高程序的重用性和扩展性，为程序提供模块化的功能，可以根据实际应用的需要编写用于特定功能的PL/SQL块。另外，使用存储过程还有利于对程序的维护和管理。这一章主要介绍存储过程的创建、调用以及参数传递方面的内容。

本章重点：

- ☐ 存储过程的创建
- ☐ 3种参数模式：IN、OUT、IN OUT
- ☐ 存储过程的调用方法
- ☐ 存储过程的删除
- ☐ 参数传递的方式

### 22.1 创建存储过程

在PL/SQL中，存储过程可以认为是一个可以执行某一个特定操作的子程序。一个存储过程包括过程说明和过程体两个部分。其中，过程体可以包括声明部分、可执行部分和异常处理部分。以关键字PROCEDURE开头的是说明部分；以关键字IS或者AS开头，以关键字END结尾的是过程体部分。创建存储过程的语法规则如下：

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(argument1[model] datatype1 [,argument2[model] datatype2...])
{IS | AS}
[local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [procedure_name];
```

其中，CREATE [OR REPLACE] PROCEDURE表示要创建一个保存在数据库中的独立过程。关键字OR REPLACE是可选的。当使用OR REPLACE关键字时，如果该过程已经存在，则首先将该过程删除，然后再重新创建一个新的存储过程。如果该存储过程不存在，则直接创建一个新的存储过程。procedure\_name表示存储过程的名字；argument1、argument2表示的是存储过程的参数；model表示指定的参数模式，参数模式包括IN、OUT、IN OUT三种，默认的参数模式为IN；datatype1、datatype2表

示参数的数据类型。存储过程中参数是可选的。一个存储过程可以没有任何参数，也可以只有一个参数，也可以含有多个参数。

关键字IS或者AS表示要开始一个过程体。存储过程中，一个过程体是一个PL/SQL语句块，这个语句块可以包含声明部分、可执行部分和异常处理部分。其中，local declarations表示声明部分。在声明部分可以对游标、常量、变量、异常或者嵌套子程序进行声明。这些声明的内容都是本地类型的，当程序退出时，这些内容就会被自动地销毁。

关键字BEGIN和EXCEPTION之间的是可执行部分。在可执行部分中，可以包含赋值语句、流程控制语句以及和Oracle数据库操作相关的语句。关键字EXCEPTION和END之间的是异常处理部分。异常处理部分是可选的，在一个存储过程中，也可以没有异常处理部分。

关键字END后面的procedure\_name表示的是存储过程的名字，这个存储过程的名字要与关键字PROCEDURE后面的procedure\_name的名字一致。关键字END后面的procedure\_name是可选的。但是在实际应用中，应该在关键字END后面将存储过程的名字标示出来。这样也有利于对程序的阅读。

**注意**

在存储过程的创建语句中，关键字IS或者AS后面是声明部分，在存储过程的声明部分中不需要使用关键字DECLARE。

在Oracle数据库中，当一个存储过程创建完成之后，首先会编译这个存储过程，编译完成之后才会把它存储到数据库中。了解了存储过程创建的语法规则，下面就来看两个创建数据库中存储过程的例子。首先看一个创建无参数的存储过程的例子。

**例22.1** 创建一个用来显示当前系统日期和时间的存储过程。

```
CREATE OR REPLACE PROCEDURE sysdata_time
AS
BEGIN
    DBMS_SESSION.SET_NLS('NLS_DATE_FORMAT','YYYY/MM/DD HH24:MI:SS');
    DBMS_OUTPUT.PUT_LINE(SYSDATE);
END sysdata_time;
```

这段PL/SQL语句是创建了一个可以显示当前系统日期和时间的存储过程。其中，sysdata\_time表示存储过程的名字。在BEGIN部分，DBMS\_SESSION是一个实现PL/SQL中ALTER SESSION命令的系统包，SET\_NLS用来设置NLS（本地化语言支持）特性。参数NLS\_DATE\_FORMAT用来更改日期时间的显示格式，参数'YYYY/MM/DD HH24:MI:SS'表示要显示的日期时间格式。DBMS\_OUTPUT.PUT\_LINE用来将当前系统日期时间更改后的格式显示输出。

**说明**

在SQL语句中可以使用ALTER SESSION SET NLS\_DATE\_FORMAT('YYYY/MM/DD HH24:MI:SS')设置日期时间显示格式，在PL/SQL中不允许直接使用ALTER SESSION命令，需要通过DBMS\_SESSION包来执行ALTER SESSION命令。

在实际应用中，大多数的时候是需要创建一个有参数的存储过程，下面就来看一个创建有参数的存储过程的例子。

**例22.2** 创建含有参数的存储过程，向学生表中插入数据记录。

```
CREATE OR REPLACE PROCEDURE insert_student(
    p_stuID      t_student.stuID%TYPE,
    p_stuName    t_student.stuName%TYPE,
```

```
p_age      t_student.age%TYPE,
p_sex      t_student.sex%TYPE,
p_birth    t_student.birth%TYPE')
AS
BEGIN
    /*插入学生信息*/
    INSERT INTO t_student
        VALUES(p_stuID, p_stuName, p_age, p_sex, p_birth);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN                -- 捕获异常
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log                      -- 将错误信息写入日志表中
            VALUES(stuID is not repeated! ,SYSDATE, 'admin');
    WHEN OTHERS THEN                          -- 捕获异常
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log                      -- 将错误信息写入日志表中
            VALUES('Oracle error!' ,SYSDATE, 'admin');
END insert_student;
```

这段PL/SQL语句是创建了一个含有参数的存储过程。在这个存储过程中，共定义了5个参数用来表示学生信息的参数，这些参数的参数模式是存储过程默认的参数模式IN。在BEGIN部分，使用INSERT INTO语句插入一条学生信息。在关键字VALUES中的值就是存储过程中定义的参数的值。EXCEPTION部分用来捕获异常，当使用INSERT INTO语句向数据库中有唯一约束条件对应的列中插入重复的值时，会发生DUP\_VAL\_ON\_INDEX异常。WHEN OTHERS THEN子句捕获所有可能发生的异常；在关键字END后面跟的是创建存储过程时的过程名insert\_student。

如果希望在另外一个PL/SQL语句块中调用这个名为insert\_student的存储过程，可以通过如下的方式调用。

```
BEGIN
    insert_student('s145203', '杜玉', 22, '女', 19870513);
END;
```

当调用insert\_student存储过程时，值's145203'就赋值给了参数p\_stuID，值'杜玉'就赋值给了参数p\_stuName，值22就赋值给了p\_age，值'女'就赋值给了p\_sex，值19870513就赋值给了p\_birth。其中，'s145203'、'杜玉'、22、'女'和19870513这些值被认为是实际参数，而存储过程中声明的参数p\_stuID、p\_stuName、p\_age、p\_sex和p\_birth这些值被认为是形式参数。有关实际参数和形式参数的详细内容可以参看22.3.2小节。

## 22.2 参数模式

在22.1节讲到的创建存储过程的语法规则中，在讲解存储过程的参数时提到了参数模式。在存储过程中通过使用参数模式来定义存储过程中参数（形式参数）的行为。形式参数模式包括3种：IN、OUT、IN OUT，默认的参数模式为IN。本节就来介绍这三种参数模式的含义及其使用方法。

### 22.2.1 IN模式

在创建存储过程时，其参数模式的默认值为IN模式。如果该存储过程中的参数模式为IN模式，那

么调用该存储过程时，实际参数的值将传递给被调用的存储过程。在该存储过程中，被指定为IN模式的参数（形式参数）的作用相当于一个常量，其值只能被读取，不能为其进行赋值操作。

存储过程中如果参数的模式为IN模式，那么其形式参数对应的实际参数既可以是一个常量、文字也可以是一个被初始化的变量，还可以是一个表达式。（有关调用带有输入参数的存储过程的内容可以参看22.3.2小节）下面这个例子中，创建一个参数模式为IN模式的存储过程，将学生的成绩信息插入到新表中。

例22.3 创建以标量变量作为输入参数的存储过程，将学生的成绩信息插入到新表中。

```
CREATE OR REPLACE PROCEDURE ins_result
(p_stuID IN t_result.stuID%TYPE,
p_curID IN t_result.curID %TYPE
)
AS
    v_result INT;                -- 定义表示学生成绩的变量
    e_illegalValue EXCEPTION;
BEGIN
    /*查询指定学生编号和课程编号的学生成绩*/
    SELECT result INTO v_result
    FROM t_result
    WHERE stuID = p_stuID
    AND curID = p_curID;
    /*当查询到的学生成绩大于100 或者小于0时，抛出e_illegalValue异常*/
    IF v_result >100 OR v_result <0 THEN
        RAISE e_illegalValue;
    ELSE
        /*将课程成绩插入到新的成绩表中*/
        INSERT INTO t_newResult
        VALUES(p_stuID,p_curID, v_result);
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('stuID is not exist!',SYSDATE, 'admin');
    WHEN e_illegalValue THEN
        DBMS_OUTPUT.PUT_LINE ('查询成绩不合法（大于100或者小于0）');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('result is illegal!',SYSDATE, 'admin');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!',SYSDATE, 'admin');
END ins_result;
```

这段PL/SQL语句是创建了一个参数模式为IN模式的存储过程ins\_result。在该存储过程中，含有两个形式参数，p\_stuID和p\_curID分别表示学生编号和课程编号。在关键字AS之后，声明了一个表示学生成绩的标量变量和一个自定义的异常e\_illegalValue。在BEGIN部分使用SELECT语句查询学生的课程成绩，如果查询的学生成绩大于100或者小于0就会抛出异常，否则就将该学生的课程成绩插入到新的



数据表t\_newResult中。在EXCEPTION部分对抛出的异常进行捕获。在关键字END后面跟的是创建函数时的过程名ins\_result。

当这个存储过程被调用时，存储过程会接收学生编号和课程编号两个变量值，然后根据这两个变量的值在t\_result对其成绩进行查询。

在创建存储过程的PL/SQL语句中，除了可以使用标量变量作为输入参数之外，也可以使用记录或者集合变量作为输入参数。

**例22.4** 创建以记录变量作为输入参数的存储过程，向学生表中插入数据记录。

```
CREATE OR REPLACE PROCEDURE ins_ student
(student_record t_student %ROWTYPE)
AS
BEGIN
    /*插入学生信息*/
    INSERT INTO t_student
    VALUES student_record;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE ('违反一致性约束');
        INSERT INTO t_log
        VALUES(stuID is not repeated!',SYSDATE, 'admin');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log
        VALUES('Oracle error!',SYSDATE, 'admin');
END ins_ student;
```

这段PL/SQL语句创建了一个向学生表中插入数据记录的存储过程ins\_ student。在这个存储过程中，使用记录变量student\_record作为存储过程ins\_ student的输入参数（这里省略了参数模式的IN关键字）。在BEGIN部分，利用记录变量student\_record向学生信息表t\_student中插入学生信息。EXCEPTION部分用来捕获异常，当使用INSERT INTO 语句向数据库中有唯一约束条件对应的列中插入重复的值时，会发生DUP\_VAL\_ON\_INDEX 异常。WHEN OTHERS THEN子句捕获所有可能发生的异常。在关键字END后面跟的是创建存储过程时的过程名ins\_ student。

集合变量也可以作为存储过程的输入参数。可以使用用户自定义的嵌套表类型或者是可变数组类型作为存储过程中的输入参数。

**例22.5** 创建以集合变量作为输入参数的存储过程，向学生表中插入数据记录。

```
CREATE OR REPLACE PROCEDURE ins_ studentTable
(stuID_table stuID_table_type,
stuName_table stuName_table_type,
age_table age_table_type,
sex_table sex_table_type,
birth_table birth_table_type
)
AS
BEGIN
    FOR i IN 1.. stuID_table.COUNT LOOP
        /*插入学生信息*/
```

```
INSERT INTO t_student
VALUES(stuID_table(I), stuName_table(I), age_table(I), sex_table(I), birth_table(I));
END LOOP;
EXCEPTION
WHEN DUP_VAL_ON_INDEX THEN                                -- 捕获异常
    DBMS_OUTPUT.PUT_LINE ('违反一致性约束');
    INSERT INTO t_log                                       -- 将错误信息写入日志表中
    VALUES(stuID is not repeated! ,SYSDATE, 'admin');
WHEN OTHERS THEN                                          -- 捕获异常
    DBMS_OUTPUT.PUT_LINE ('发生其他错误');
    INSERT INTO t_log                                       -- 将错误信息写入日志表中
    VALUES('Oracle error!' ,SYSDATE, 'admin');
END ins_studentTable;
```

这段PL/SQL语句创建了一个向学生表中插入数据记录的存储过程ins\_studentTable。在这个存储过程中，使用嵌套表作为存储过程ins\_studentTable的输入参数（这里省略了参数模式的IN关键字）。在BEGIN部分，通过使用FOR循环语句将嵌套表中的数据记录一一取出并插入到学生信息表t\_student中。EXCEPTION部分用来捕获异常，当使用INSERT INTO 语句向数据库中有唯一约束条件对应的列中插入重复的值时，会发生DUP\_VAL\_ON\_INDEX 异常。WHEN OTHERS THEN子句捕获所有可能发生的异常。在关键字END后面跟的是创建存储过程时的过程名ins\_studentTable。

当使用PL/SQL嵌套表作为数据表中数据列的数据类型时，首先需要使用CREATE TYPE命令创建一个嵌套表类型，而且还要在创建数据表时为以嵌套表作为数据类型的列指定一个存储表。所以在使用嵌套表作为输入参数之前，还需要分别创建嵌套表类型。

```
CREATE TYPE stuID_table_type IS TABLE OF VARCHAR(15);
CREATE TYPE stuName_table_type IS TABLE OF VARCHAR(10);
CREATE TYPE age_table_type IS TABLE OF INT;
CREATE TYPE sex_table_type IS TABLE OF VARCHAR(2);
CREATE TYPE birth_table_type S TABLE OF DATETIME;
```

### 22.2.2 OUT模式

如果在创建存储过程时，存储过程中的参数模式指定为OUT模式，那么在完成存储过程的调用后，实际参数的值将返回给存储过程的调用者。在该存储过程中，被指定为OUT模式的参数（形式参数）的作用相当于一个变量，该形式参数必须被赋值。其值既可以被读取，也可以被写入。可以把它当做本地变量来使用。

**说明** 如果存储过程中参数的模式定义为OUT，那么在该存储过程调用之前，其参数可以有值存在。当存储过程被调用时，该值就会被忽略。除非使用了NOCOPY编译器提示进行参数传递或是因为存储过程中有未处理的异常而退出。

存储过程中如果参数的模式为OUT模式，则其形式参数会被初始化为NULL值，其形式参数对应的实际参数必须是变量，不可以是常量或者表达式。（有关调用带有输出参数的存储过程的内容可以参看22.3.3小节）下面这个例子中，创建一个参数模式为OUT模式的存储过程，将原来没有津贴的教师增加津贴100。

**例22.6** 创建以标量变量作为输出参数的存储过程，将原来没有津贴的教师增加津贴100。

```
CREATE OR REPLACE PROCEDURE update_pension
(p_teaID t_teacher. teaID%TYPE,      -- 教师编号
pension OUT REAL                     -- 教师津贴
)
AS
BEGIN
    /* 查询指定教师编号的教师津贴*/
    SELECT pension INTO pension
    FROM t_teacher
    WHERE teaID= p_teaID;
    /*如果教师津贴为NULL*/
    IF pension IS NULL THEN
        /*修改教师津贴*/
        pension:= pension + 100;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('teaID is not exist!',SYSDATE, 'admin');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!',SYSDATE, 'admin');
END update_pension;
```

这段PL/SQL语句是创建了一个参数模式为OUT模式的存储过程update\_pension。在该存储过程中，含有两个形式参数，p\_teaID和pension分别表示教师编号和教师津贴。其中，p\_teaID参数模式为IN，pension的参数模式为OUT。在BEGIN部分使用SELECT语句查询指定教师编号的教师津贴信息，如果教师津贴为NULL就将该教师的津贴增加100。在EXCEPTION部分对抛出的异常进行捕获。

当这个存储过程被调用时，其实际参数的值就会丢失。当完成存储过程的调用之后，形式参数的值就被赋值给实际参数。

**注意** 存储过程中如果参数的模式为OUT模式，则被定义为OUT模式的形式参数的数据类型是不能有NOT NULL约束的。如果将一个有NOT NULL约束的数据指定为OUT模式的形式参数，则会抛出VALUE\_ERROR异常。

在创建存储过程的PL/SQL语句中，除了可以使用标量变量作为输出参数之外，也可以使用记录或者集合变量作为输出参数。

**例22.7** 创建以记录变量作为输出参数的存储过程，查询教师记录。

```
CREATE OR REPLACE PROCEDURE select_teacher
(p_teaID t_teacher. teaID %TYPE,
p_teacher_record OUT t_teacher %ROWTYPE
)
AS
```

```
BEGIN
    /*查询教师信息*/
    SELECT teaName,dept ,profession INTO p_teacher _ record
    FROM t_eacher
    WHERE teaID = p_ teaID;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('teaID is not exsit!' ,SYSDATE, 'admin');
    WHEN OTHERS THEN              -- 捕获异常
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!' ,SYSDATE, 'admin');
END select _ teacher;
```

这段PL/SQL语句创建了一个查询教师记录的存储过程select\_teacher。在这个存储过程中，使用记录变量p\_teacher\_record作为存储过程select\_teacher的输出参数，p\_teaID作为输入参数。在BEGIN部分，将查询到的教师记录放入到记录变量p\_teacher\_record中。EXCEPTION部分用来捕获异常，当没有查询到数据时，会发生NO\_DATA\_FOUND异常。WHEN OTHERS THEN子句捕获所有可能发生的异常。在关键字END后面跟的是创建存储过程时的过程名select\_teacher。

集合变量也可以作为存储过程的输出参数。可以使用用户自定义的嵌套表类型或者是可变数组类型作为存储过程中的输出参数。

**例22.8** 创建以集合变量作为输出参数的存储过程，查询教师记录。

```
CREATE OR REPLACE PROCEDURE select_ teacherTable
(p_ deptID t_eacher. deptID %TYPE,
teaID_table OUT teaName _table_type,
teaName _table OUT teaName _table_type,
dept _table OUT dept _table_type,
profession _ table OUT profession _table_type,
)
AS
BEGIN
    /*查询教师信息*/
    SELECT teaID ,teaName,dept ,profession
        INTO teaID_table ,teaName _table, dept_table, profession_table
    FROM t_eacher
    WHERE deptID = p_deptID;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('teaID is not exsit!' ,SYSDATE, 'admin');
    WHEN OTHERS THEN              -- 捕获异常
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!' ,SYSDATE, 'admin');
```



```
END select_teacherTable;
```

这段PL/SQL语句创建了一个查询教师记录的存储过程select\_teacherTable。在这个存储过程中，使用嵌套表作为存储过程select\_teacherTable的输出参数。在BEGIN部分，将查询到的教师记录分别放入到嵌套表teaID\_table、teaName\_table、dept\_table和profession\_table中。EXCEPTION部分用来捕获异常，当没有查询到数据时，会发生NO\_DATA\_FOUND异常。WHEN OTHERS THEN子句捕获所有可能发生的异常。在关键字END后面跟的是创建存储过程时的过程名select\_teacherTable。

当使用PL/SQL嵌套表作为数据表中数据列的数据类型时，首先需要使用CREATE TYPE命令创建一个嵌套表类型，而且还要在创建数据表时为以嵌套表作为数据类型的列指定一个存储表。所以在使用嵌套表作为输出参数之前，还需要分别创建嵌套表类型。

```
CREATE TYPE teaID_table_type IS TABLE OF VARCHAR(15);
CREATE TYPE teaName_table_type IS TABLE OF VARCHAR(10);
CREATE TYPE dept_table_type IS TABLE OF VARCHAR(20);
CREATE TYPE profession_table_type IS TABLE OF VARCHAR(10);
```

### 22.2.3 IN OUT模式

如果在创建存储过程时，存储过程中的参数模式指定为IN OUT模式，那么它会向存储过程传递初始值，同时也会向调用者返回更新后的结果值。即在调用该存储过程时，实际参数的值将传递给被调用的存储过程；在完成存储过程的调用后，实际参数的值将返回给存储过程的调用者。在该存储过程中，被指定为IN OUT模式的参数（形式参数）的作用相当于一个已初始化的变量，其值既可以被读取，也可以被写入。

存储过程中如果参数的模式为IN OUT模式，其形式参数对应的实际参数必须是变量，不可以是常量或者表达式。（有关调用带有输入输出参数的存储过程的内容可以参看22.3.4小节）下面这个例子中，创建一个参数模式为IN OUT模式的存储过程，计算两个数相除的余数。

**例22.9** 计算两个数相除的余数。

```
CREATE OR REPLACE PROCEDURE divide_mod
(p_num1 IN OUT NUMBER,
p_num2 IN OUT NUMBER
)
AS
    v_trunc NUMBER;           -- 表示除法结果的商
    v_mod NUMBER;            -- 表示除法结果的余数
BEGIN
    v_trunc := TRUNC(p_num1/p_num2);
    v_mod := MOD(p_num1, p_num2);
    p_num1 := v_trunc;
    p_num2 := v_mod;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE ('除数值不能为0');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
END divide_mod;
```



这段PL/SQL语句是创建了一个参数模式为IN OUT模式的存储过程divide\_mod。在该存储过程中，含有两个形式参数，p\_num1和p\_num2分别表示被除数和除数。它们的参数模式都为IN OUT。关键字AS之后定义了两个NUMBER类型的变量，分别用来表示除法结果的商和余数。在BEGIN部分计算p\_num1和p\_num2相除的结果，并将得到的商赋值给p\_num1，将得到的余数赋值给p\_num2。其中，函数TRUNC表示截取数字，函数MOD表示求余。（有关这两个函数的具体用法可以参看10.2节）。在EXCEPTION部分对抛出的异常进行捕获。

## 22.3 调用存储过程

当存储过程创建完成之后，就可以调用该存储过程了。在PL/SQL中，可以直接对存储过程进行调用。由于创建的存储过程可以是无参数的，也可以是有参数的，所以存储过程的调用方法也有所不同。本节就来介绍有参数和无参数的情况下存储过程的调用方法。

### 22.3.1 调用无参数的存储过程

在PL/SQL中，如果该存储过程没有参数，可以直接通过存储过程名对其进行调用。例如，对于例22.1中显示当前系统日期和时间的存储过程，在PL/SQL语句块中就可以使用下面的方法对其进行调用。

```
BEGIN
    sysdata_time;
END;
```

在这个PL/SQL语句块中只包含一个可执行部分BEGIN。其中，sysdata\_time就是存储过程的名字。最后以关键字END结束。其显示的结果如下所示。

```
2009/09/23 21:23:40
```

在SQL\*Plus中，需要使用EXEC命令或者是CALL命令调用存储过程。如果使用SQL语句，在SQL\*Plus中使用EXEC命令调用例22.1中显示当前系统日期和时间的存储过程方法如下所示。

```
EXEC sysdata_time
```

这里sysdata\_time表示存储过程的名字，对于无参数的存储过程使用EXEC命令直接引用其过程名即可完成对该存储过程的调用。

如果使用SQL语句，在SQL\*Plus中就可以使用CALL命令调用例22.1中显示当前系统日期和时间的存储过程，方法如下所示。

```
CALL sysdata_time()
```

这里sysdata\_time表示存储过程的名字。当使用CALL命令调用存储过程时，即使该存储过程没有参数，其后面的括号也是必需的。

### 22.3.2 调用带有输入参数的存储过程

在一个存储过程中，如果该存储过程中带有输入参数，那么在调用该存储过程时，需要为存储过程中的输入参数提供数据值。例如，如果调用例22.3中将学生的成绩信息插入到新的数据表的存储过程ins\_result，可以使用下面的方法完成。

```
BEGIN
```

```
ins_result ('s102203', 't105');  
END;
```

在这个PL/SQL语句块中只包含一个可执行部分BEGIN。其中，ins\_result是存储过程的名字。在该存储过程名之后的括号中包含有两个参数。值s102203表示学生编号，该值被赋值给了存储过程中的第一个参数p\_stuID；值t105是课程编号，该值会赋值给存储过程中的第二个参数p\_curID。最后以关键字END结束。

这里，介绍两个概念：实际参数和形式参数。在PL/SQL中，调用存储过程中的参数列表中所引用的变量或者表达式称为实际参数（以后称为实参）。在存储过程中声明的变量称为形式参数（以后称为形参）。对于用例22.3中将学生的成绩信息插入到新的数据表的这个存储过程来说，值s102203和值t105就是实际参数，在存储过程ins\_result中声明的变量p\_stuID和p\_curID为形式参数。在调用存储过程时，PL/SQL会将实参的值赋值给形参，并对其进行必要的类型转换。

**注意** 实参和形参的数据类型必须相互匹配。如果形参和实参的数据类型不匹配，会引发VALUE\_ERROR异常。

在PL/SQL语句中，记录或者集合变量也可以作为存储过程的输入参数，因此也可以在PL/SQL语句块中对以记录或者集合变量作为输入参数的存储过程进行调用。

例如，对于例22.4中的以记录变量作为输入参数的存储过程ins\_student，在PL/SQL语句块中就可以使用下面的方法对其进行调用。

```
DECLARE  
    student_record t_student %ROWTYPE;  
BEGIN  
    student_record.stuID = 's131203';  
    student_record.stuName = '彭庄';  
    student_record.age = 23;  
    student_record.sex = '男';  
    student_record.birth = 19860212;  
    ins_student(student_record);      -- 调用存储过程ins_student  
END;
```

在这个PL/SQL语句块中，DECLARE部分声明了一个记录类型的变量。在BEGIN部分为该记录变量中的每一个记录成员赋值，然后调用ins\_student存储过程，并将student\_record作为参数传递给该存储过程。

对于例22.5中的以嵌套表变量作为输入参数的存储过程ins\_studentTable，在PL/SQL语句块中就可以使用下面的方法对其进行调用。

```
DECLARE  
    stuID_table stuID_table_type := stuID_table_type('s131204', 's131205');  
    stuName_table stuName_table_type := stuName_table_type('张三', '李四');  
    age_table age_table_type := age_table_type(21, 21);  
    sex_table sex_table_type := sex_table_type('男', '女');  
    birth_table birth_table_type := birth_table_type(19880313, 19880218);  
BEGIN  
    ins_studentTable (stuID_table, stuName_table, age_table, sex_table, birth_table); -- 调用存储过程  
END;
```

## 零基础学SQL

在这个PL/SQL语句块中，DECLARE部分对每一个嵌套表都进行赋值操作，在BEGIN部分调用ins\_studentTable存储过程，并将已经赋值好的嵌套表作为参数传递给该存储过程。

### 22.3.3 调用带有输出参数的存储过程

在一个存储过程中，如果该存储过程中带有输出参数，那么在完成对该存储过程的调用之后，存储过程中指定为输出参数的形参的值会赋值给实参。例如，如果调用例22.6中将原来没有津贴的教师津贴增加100的这个存储过程，可以使用下面的方法完成。

```
DECLARE
    v_pension t_teacher. pension %TYPE,
BEGIN
    update_pension ('t156354', v_pension);
    DBMS_OUTPUT.PUT_LINE ('v_pension的值为 ' || v_pension);
END;
```

在这个PL/SQL语句块中，DECLARE部分首先声明了一个表示教师津贴的变量v\_pension，在BEGIN部分调用了存储过程update\_pension，并为其中的输入参数p\_teaID赋值为't156354'，当存储过程调用结束之后，PL/SQL就会把修改后教师津贴的值赋给实参v\_pension。其显示结果如下：

```
v_pension的值为 100
```

#### 注意

存储过程中如果参数为OUT模式，其与其形参对应的实参必须是一个变量；不能是一个常量，也不能是一个表达式。例如，下面的存储过程在调用的时候会出现错误。

```
update_pension ('t156354', v_pension+100); -- 存储过程调用不合法
```

在PL/SQL语句中，记录或者集合变量也可以作为存储过程的输出参数，因此也可以在PL/SQL语句块中对以记录或者集合变量作为输出参数的存储过程进行调用。

例如，对于例22.7中的以记录变量作为输出参数的存储过程select\_teacher，在PL/SQL语句块中就可以使用下面的方法对其进行调用。

```
DECLARE
    teacher_record t_teacher %ROWTYPE;
BEGIN
    select_teacher ('t156354', teacher_record); -- 调用存储过程select_teacher
    DBMS_OUTPUT.PUT_LINE ('教师姓名为: ' || teacher_record. teaName);
    DBMS_OUTPUT.PUT_LINE ('教师所在院系为: ' || teacher_record. dept);
    DBMS_OUTPUT.PUT_LINE ('教师职称为: ' || teacher_record. profession);
END;
```

在这个PL/SQL语句块中，DECLARE部分声明了一个记录类型的变量teacher\_record。在BEGIN部分调用select\_teacher存储过程，由于记录变量teacher\_record在存储过程select\_teacher中的参数模式被定义为OUT，因此在存储过程调用结束之后，会将查询得到的数据传递给存储过程的调用者。在teacher\_record记录中就存储了所要查询的教师信息。其显示结果如下。

```
教师姓名为: 王新
教师所在院系为: 数学系
教师职称为: 讲师
```

对于例22.8中的以嵌套表变量作为输出参数的存储过程select\_teacherTable，在PL/SQL语句块中就可以使用下面的方法对其进行调用。

```
DECLARE
    teaID_table teaID_table_type;
    teaName_table teaName_table_type;
    dept_table dept_table_type;
    profession_table profession_table_type;
BEGIN
    select_teacherTable ('t_15', teaID_table, teaName_table, dept_table, profession_table); -- 调用存储过程
    FOR I IN 1.. teaID_table.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE ('教师编号为: ' || teaID_table(i));
        DBMS_OUTPUT.PUT_LINE ('教师姓名为: ' || teaName_table(i));
        DBMS_OUTPUT.PUT_LINE ('教师所在院系为: ' || dept_table(i));
        DBMS_OUTPUT.PUT_LINE ('教师职称为: ' || profession_table(i));
    END LOOP;
END;
```

在这个PL/SQL语句块中，DECLARE部分声明四个嵌套表类型。在BEGIN部分调用select\_teacherTable存储过程，由于嵌套表teaID\_table、teaName\_table、dept\_table和profession\_table在存储过程select\_teacherTable中的参数模式被定义为OUT，因此在存储过程调用结束之后，会将查询得到的数据传递给存储过程的调用者。在嵌套表teaID\_table、teaName\_table、dept\_table和profession\_table中就存储了所要查询的教师信息。其显示结果如下：

```
教师编号为: t156354
教师姓名为: 王新
教师所在院系为: 数学系
教师职称为: 讲师
教师编号为: t156355
教师姓名为: 李中
教师所在院系为: 数学系
教师职称为: 教授
```

### 22.3.4 调用带有输入输出参数的存储过程

在一个存储过程中，如果该存储过程中带有输入输出参数，那么在调用该存储过程时，实参的值将传递给被调用的存储过程；在完成存储过程的调用后，实参的值将返回给存储过程的调用者。例如，如果调用例22.9中计算两个数相除的余数的这个存储过程，可以使用下面的方法完成。

```
DECLARE
    n1:=13;
    n2:=5;
BEGIN
    divide_mod(n1,n2);
    DBMS_OUTPUT.PUT_LINE ('13除5的商为: ' || n1);
    DBMS_OUTPUT.PUT_LINE ('13除5的余数为: ' || n2);
END;
```

在这个PL/SQL语句块中，DECLARE部分定义了两个变量n1和n2，并为它们分别赋初值。在BEGIN部分调用了存储过程divide\_mod，并将调用存储过程后计算的结果显示输出。



## 零基础学SQL

13除5的商为：2

13除5的余数为：3

**注意** 存储过程中如果参数为IN OUT模式，其与其形参对应的实参必须是一个变量；不能是一个常量，也不能是一个表达式。例如，下面的存储过程在调用的时候会出现错误。

```
divide_mod(n1+1,n2+3); -- 存储过程调用不合法
```

## 22.4 参数传递

存储过程通过使用参数传递信息。PL/SQL中参数传递有两种方式，一种是引用传递，一种是值传递。引用传递是将实参的指针传递给对应的形参中，而值传递是指将实参的值拷贝给了对应的形参。PL/SQL中，参数模式被指定为IN模式的是通过引用传递来传递参数，参数模式被指定为OUT或者IN OUT模式的是通过值传递来传递参数。在调用存储过程时，既可以使用参数名称也可以使用参数位置来为实参提供数据。通过参数位置或者参数名称可以把实参和形参关联起来。这一节就来介绍有关存储过程参数传递的内容。

### 22.4.1 使用参数位置传递参数值

在PL/SQL中，可以使用参数位置传递参数变量或者数据。通过使用参数位置传递变量或者数据时，PL/SQL编译器会将第一个实参的值与第一个形参的值相关联，第二个实参的值与第二个形参的值相关联，按照参数定义的顺序依次为形参赋值。例如，对于例22.3中将学生的成绩信息插入到新的数据表的这个存储过程ins\_result，通过参数位置传递，可以使用下面的方法对其进行调用。

```
DECLARE
    v_stuID VARCHAR2(15)
    v_curID VARCHAR2(15)
BEGIN
    ins_result (v_stuID, v_curID);
END;
```

### 22.4.2 使用参数名称传递参数值

在PL/SQL中，可以使用参数名称传递参数变量或者数据。可以通过使用关联参照符=>将实参和形参关联起来。例如，对于例22.3中将学生的成绩信息插入到新的数据表的这个存储过程ins\_result，通过参数名称传递，可以使用下面的方法对其进行调用。

```
DECLARE
    v_stuID VARCHAR2(15);
    v_curID VARCHAR2(15);
BEGIN
    ins_result (p_stuID=> v_stuID, p_curID=> v_curID);
END;
```

这个PL/SQL语句块中，通过使用参数名称为参数传递变量。在DECLARE部分声明了两个变量v\_stuID和v\_curID，分别用来表示学生编号和课程编号。在BEGIN部分，调用ins\_result存储过程，并通



过关联参照符=>为存储过程中的参数提供变量。

在PL/SQL中，使用参数名称传递参数变量或者数据时，无须知道形参在参数列表中的顺序，可以对参数的位置重新整理。

```
DECLARE
    v_stuID VARCHAR2(15);
    v_curID VARCHAR2(15);
BEGIN
    ins_result ( p_curID=> v_curID ,p_stuID=> v_stuID);
END;
```

### 22.4.3 使用位置和名称传递参数值

在PL/SQL中，也可以混合使用参数名称和参数位置来传递参数变量或者数据。在混合使用参数名称和参数位置来传递参数变量或者数据时，参数位置传递参数值的方法需要在参数名称传递参数值的方法之前使用。例如，对于例22.3中将学生的成绩信息插入到新的数据表的这个存储过程ins\_result，混合使用参数名称和参数位置传递，可以使用下面的方法对其进行调用。

```
DECLARE
    v_stuID VARCHAR2(15);
    v_curID VARCHAR2(15);
BEGIN
    ins_result (v_curID ,p_stuID=> v_stuID);
END;
```

这个PL/SQL语句块中，调用存储过程ins\_result时混合使用参数名称和参数位置传递两种传递方法。其中，第一个参数v\_curID是按参数位置传递，第二个参数是按参数名称传递。如果将第一个参数和第二个参数传递的方法反过来，那么在调用ins\_result存储过程时就会出现错误。例如，下面的存储过程的调用是不合法的。

```
DECLARE
    v_stuID VARCHAR2(15);
    v_curID VARCHAR2(15);
BEGIN
    ins_result (p_stuID=> v_stuID ,v_curID);      -- 存储过程调用不合法
END;
```

### 22.4.4 使用NOCOPY编译提示传递参数

在PL/SQL中，传递方式有两种：值传递和引用传递。当使用引用传递时，是将实参的指针传递给对应的形参中，因此使用引用传递会使程序的执行速度更快，而值传递是将实参的值拷贝给了对应的形参。对于使用集合或者记录这样的数据结构作为参数来说，如果使用值传递，会使程序的执行速度变慢。那么对于参数模式定义为OUT或者IN OUT模式的集合或者记录来说，有没有什么办法可以让它通过引用的方式传递，从而加快程序的执行效率呢？这就需要使NOCOPY编译提示。使用NOCOPY编译提示声明一个参数的语法规则如下：

```
argument[model] NOCOPY datatype
```

其中，argument表示参数的名字；model表示指定的参数模式，参数模式包括IN、OUT、IN OUT

## 零基础学SQL

三种，默认的参数模式为IN，datatype表示参数的数据类型。

在PL/SQL中，如果一个参数使用了NOCOPY编译提示，那么编译器会试图按照引用的方式来传递OUT模式或者IN OUT模式的参数。

```
CREATE OR REPLACE PROCEDURE procedurc_nocopy
(p_teaID IN t_teacher. teaID%TYPE,          -- 教师编号
p_salary OUT NOCOPY REAL,                    -- 教师工资
p_pension IN OUT NOCOPY REAL                 -- 教师津贴
)
AS
-- 过程体代码
```

在这个PL/SQL片段中，在存储过程procedurc\_nocopy中有3个参数。其中，参数p\_teaID是IN模式的，参数p\_salary是OUT模式的，参数p\_pension是IN OUT模式的。这里使用NOCOPY编译提示声明了参数p\_salary和p\_pension。这样，对于参数p\_salary和p\_pension来说，编译器将会试图按照引用的方式来传递这两个参数。

这里需要说明的一点是，试图按照引用的方式来传递参数p\_salary和p\_pension。即虽然在一般情况下，使用NOCOPY编译提示编译器会将OUT模式或者IN OUT模式的参数按照引用的方式进行传递，但是也有可能编译器不采用引用方式，还采用原来的值传递的方式对参数p\_salary和p\_pension进行传递。这是因为NOCOPY只是一个编译提示，它并不是一个指令。

**注意** NOCOPY编译提示只能用在OUT模式或者IN OUT模式的参数上，不能用在IN模式的参数上。如果将NOCOPY编译提示使用到IN模式的参数上，会得到一个编译错误。

在PL/SQL中，如果模式为OUT或者IN OUT的参数为集合或者记录，那么使用NOCOPY编译提示可以大大提高程序的执行速度和运行性能。例如，下面的PL/SQL片段中就通过使用NOCOPY编译提示声明一个嵌套表类型的IN OUT模式的参数。

```
DECLARE
    TYPE int_array_type IS VARRAY(100) OF NUMBER;
    PROCEDURE reorganize (int_array IN OUT NOCOPY int_array_type);
AS
BEGIN
    --执行代码
END;
```

使用NOCOPY编译提示可以提高程序的执行速度和运行性能。在下面的一些情况下，不会产生错误，PL/SQL编译器会忽略NOCOPY编译提示，参数的传递方式还将是按值传递的。

- ☐ 实参受精度、NOT NULL约束条件的约束。该限制不适用长度受限的字符参数。
- ☐ 实参是索引表中的一个成员。该限制不适用实参是整个索引表的情况。
- ☐ 实参在传递时，需要进行隐式数据类型转换。
- ☐ 实参和形参都是记录，实参作为被隐式声明的游标FOR循环的索引，或者其中一个或两个使用了%ROWTYPE或%TYPE声明，而且记录中对应的列的约束不相同。
- ☐ 被远程服务器的数据库连接，进行远程过程调用的情况下，不能使用NOCOPY编译提示按引用方式传递。

### 22.4.5 使用参数的默认值

在PL/SQL中，还可以为IN模式的参数指定一个初始化的默认值，调用存储过程时，就可以不为该参数其他数据。设置参数默认值的语法规则如下：

```
argument[model] datatype {DEFAULT|:=} default_value;
```

其中，argument表示的是存储过程的参数；model表示指定的参数模式；参数模式包括IN、OUT、IN OUT三种；datatype表示参数的数据类型；关键字DEFAULT或者是“:=”表示为参数设置一个默认值；default\_value是为参数指定的默认值。

```
CREATE OR REPLACE PROCEDURE insert_teacher (  
    p_teaID      t_teacher.teaID%TYPE,  
    p_teaName    t_teacher.teaName%TYPE,  
    p_age        t_teacher.age%TYPE,  
    p_sex        t_teacher.sex%TYPE,  
    p_deptID     t_teacher.deptID %TYPE DEFAULT 't_10',  
    p_dept       t_teacher.dept %TYPE DEFAULT '计算机系',  
    p_profession t_teacher.profession %TYPE,  
    p_salary     t_teacher.salary %TYPE,  
    p_pension    t_teacher.pension %TYPE)  
AS  
BEGIN  
    /*插入教师信息*/  
    INSERT INTO t_teacher  
    VALUES(p_teaID, p_teaName, p_age, p_sex, p_deptID, p_dept, p_profession, p_salary , p_pension);  
EXCEPTION  
    WHEN DUP_VAL_ON_INDEX THEN -- 捕获异常  
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');  
        INSERT INTO t_log -- 将错误信息写入日志表中  
        VALUES(teaID is not repeated! ,SYSDATE, 'admin');  
    WHEN OTHERS THEN -- 捕获异常  
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');  
        INSERT INTO t_log -- 将错误信息写入日志表中  
        VALUES('Oracle error!' ,SYSDATE, 'admin');  
END insert_teacher;
```

在这段PL/SQL语句中，创建了一个存储过程insert\_teacher，并为其中的两个参数p\_deptID和p\_dept设置了默认值。在BEGIN部分，使用INSERT INTO语句向教师信息表中插入一条教师信息。EXCEPTION部分用来捕获并处理异常。对于该存储过程，可以使用下面的方法进行调用。

```
BEGIN  
    insert_teacher ('t105678','张一',40,'男', p_profession =>'教授', p_salary =>3800, p_pension =>300);  
END;
```

在这个存储过程的调用中，可以看到，并没有给参数p\_deptID和p\_dept赋值。如果在过程调用中并没有传入与其对应的实参值，则其对应的形参就会使用存储过程定义时的默认值。

如果在存储过程的调用中，insert\_teacher存储过程的调用接受9个实参的值，那么设置为默认值的参数的值将被实参的值所覆盖。

```
BEGIN
```

## 零基础学SQL

```
insert_teacher ('t154567','李蓝',36,'女','t_15','数学系','副教授', 3200, 100);  
END;
```

此时，存储过程中设置的默认值t\_10和计算机系将会被参数值t\_15和数学系覆盖。如果想覆盖默认参数值，一般情况下使用参数位置的方式覆盖默认值。但是在过程调用中，试图通过使用一个占位符的方式跳过其对应的形参是不允许的。

```
BEGIN  
insert_teacher ('t154567','李蓝',36,'女', , '副教授', 3200, 100); -- 过程调用不合法  
END;
```

在实际应用中，可能会出现向数据表中的某一列插入空值的情况。如果想在调用存储过程时，为其中的某一列赋空值，可以在创建存储过程时，就为其参数指定默认值为NULL，也可以使用参数位置的方式为其赋值。例如，如果想为教师信息表中教师津贴这一列赋值为空，可以使用下面的语句。

```
BEGIN  
insert_teacher ('t105678','张一',40,'男', p_profession =>'教授', p_salary =>3800, p_pension =>NULL);  
END;
```

## 22.5 删除存储过程

与在SQL语句中删除一个数据表类似，存储过程也可以删除。使用DROP命令可以将存储过程从数据字典中删除。删除存储过程的语法格式如下：

```
DROP procedure_name ;
```

其中，procedure\_name为要删除的存储过程的名字，这个存储过程一定是一个已经存在的存储过程。例如，要删除存储过程insert\_student，就可以使用下面的语句来完成。

```
DROP insert_student;
```

在执行该DROP命令时，程序会隐式执行一个COMMIT命令。如果存储过程不存在，则在使用DROP命令时会引发“Object does not exist”。

## 22.6 小结

本章主要介绍了存储过程的创建、调用以及参数传递等内容。在这一章中主要讨论了存储过程的创建过程以及调用的方法，并介绍了存储过程中形参的3种模式（IN、OUT、IN OUT）及其各自的使用方法。其中，IN模式默认的是通过引用传递来传递参数，OUT或者IN OUT模式默认的是通过值传递来传递参数。在本章的最后介绍了使用参数位置、参数名称、NOCOPY编译提示以及参数默认值实现参数传递的方法。最后介绍了如何删除一个存储过程。在第23章中，将介绍子程序中另外一个类型：函数。

## 第23章 函 数

第22章中介绍了存储过程，存储过程一般用于执行一个操作。这一章将介绍PL/SQL中子程序的另一个类型——函数。函数是一个能够返回计算结果值的子程序。函数的主要作用就是返回一个特定的结果值。通过创建和使用函数，不仅可以提高程序的重用性和扩展性，还有利于对程序的维护和管理。这一章主要介绍函数的创建、调用、参数传递以及在SQL语句中如何调用函数等内容。最后，还将对存储过程和函数之间的异同进行比较。

本章重点：

- ☐ 函数的创建
- ☐ 函数的调用
- ☐ 参数传递的方法
- ☐ 函数在SQL语句中的应用
- ☐ 函数的删除
- ☐ 存储过程与函数的比较

### 23.1 创建函数

在PL/SQL中，函数用来返回特定的结果值。一个函数包括函数说明和函数体两个部分。其中，函数体可以包括声明部分、可执行部分和异常处理部分。本节就来介绍函数创建的语法规则以及如何创建无参数和有参数的函数。

#### 23.1.1 创建函数的语法规则

使用CREATE [OR REPLACE] FUNCTION关键字可以创建一个函数。以关键字FUNCTION开头的是说明部分；以关键字IS或者AS开头，以关键字END结尾的是函数体部分。创建存储函数的语法规则如下：

```
CREATE [OR REPLACE] FUNCTION function_name
[(argument1[model] datatype1 [,argument2[model] datatype2...])
RETURN return_datatype
{IS | AS}
[local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [function_name];
```



其中，CREATE [OR REPLACE] FUNCTION表示要创建一个存储在数据库中的函数。关键字OR REPLACE是可选的。argument1、argument2表示的是函数的参数；model表示指定的参数模式，参数模式包括IN、OUT、INOUT三种，默认的参数模式为IN（有关参数模式的内容可以参看22.2节）；datatype1、datatype2表示参数的数据类型。函数中参数是可选的。一个函数可以没有任何参数，也可以只有一个参数，也可以含有多个参数。RETURN关键字后面的return\_datatype表示指定该函数返回值的类型。

关键字IS 或者AS表示要开始一个函数体。函数中，一个函数体是一个PL/SQL语句块，这个语句块可以包含声明部分、可执行部分和异常处理部分。其中，local declarations表示声明部分。在声明部分可以对游标、常量、变量、异常或者嵌套子程序进行声明。这些声明的内容都是本地类型的，当程序退出时，这些内容就会被自动地销毁。

关键字BEGIN和EXCEPTION之间的是可执行部分。在可执行部分中，可以包含赋值语句、流程控制语句以及与Oracle数据操作相关的语句。关键字EXCEPTION和END之间的是异常处理部分。异常处理部分是可选的，在一个函数中，也可以没有异常处理部分。

关键字END后面的function\_name表示的是函数的名字，这个函数的名字要与关键字FUNCTION后面的function\_name的名字一致。关键字END后面的function\_name是可选的。但是在实际应用中，应该在关键字END后面将函数的名字标示出来。这样也有利于对程序的阅读。

**注意** 在函数的创建语句中，RETURN子句表示指定该函数返回值的类型。在函数体中至少应该含有一条RETURN语句用来返回一个特定的数据值（有关函数体内的RETURN子句的详细说明可以参看23.1.2小节）。

在了解了函数创建的语法规则后，就可以应用创建函数的语法规则创建函数了。下面来看一个创建无参数的函数的例子。

**例23.1** 创建一个用来显示当前系统日期和时间的函数。

```
CREATE OR REPLACE FUNCTION sysdata_time
RETURN VARCHAR2
AS
BEGIN
    RETURN TO _CHAR(SYSDATE, 'YYYY-MM-DD HH24:MI:SS');
END sysdata_time;
```

这段PL/SQL语句是创建了一个可以显示当前系统日期和时间的函数。其中，sysdata\_time 表示函数的名字。在函数声明中，RETURN子句指定返回的数据类型为VARCHAR2类型。在BEGIN部分，使用函数TO\_CHAR将系统时间转换为以“YYYY-MM-DD HH24:MI:SS”形式显示的字符串类型，并使用RETURN子句将转换后的系统时间返回。

### 23.1.2 函数体中的RETURN子句

在上一节中，介绍了创建函数的语法规则。可以看到，在函数的创建中需要使用一个RETURN子句，其余的语法规则与创建函数的语法规则是非常相似。正如在23.1.1小节中最后在注意中提到的，除了在函数声明中需要使用RETURN子句用来指定函数返回的数据类型之外，在函数体内还需要使用一个RETURN子句返回一个特定的数据值。在函数体内RETURN子句的语法规则如下：

```
RETURN exception;
```

其中，exception是一个表达式，用来表示函数要返回的数据值。当程序执行到函数体内的RETURN语句后，程序会将其控制权立即返回给调用者。执行函数调用之后的语句。如果exception返回的数据值的数据类型与函数声明中的RETURN子句指定的返回值的数据类型不相符，则PL/SQL会将其转换为RETURN子句指定的返回值的数据类型。

在一个函数中可以有一个或者多个RETURN子句，每一个RETURN子句必须包含一个表达式，在该RETURN子句被执行时，PL/SQL程序计算该表达式的值。但是PL/SQL只会执行其中的一个RETURN子句。当其中有一个RETURN子句被执行时，就会立即结束当前的函数，并将程序控制权返回给函数的调用者。

这里需要说明的一点是，在存储过程中也可以使用RETURN子句，但是存储过程的RETURN子句没有返回值。在存储过程中的RETURN子句的作用只是将程序的控制权交给其调用环境，并不会返回任何的值或者表达式。

### 23.1.3 创建有输入参数的函数

在创建函数时，其参数模式的默认值为IN模式。如果该函数中的参数模式为IN模式，那么调用该函数时，实际参数的值将传递给被调用的函数。在该函数中，被指定为IN模式的参数的作用相当于一个常量，其值只能被读取，不能为其进行赋值操作。下面来看一个创建有输入参数的函数的例子。

例23.2 创建有输入参数的函数，返回学生的课程成绩。

```
CREATE OR REPLACE FUNCTION get_result
(p_stuID IN t_t_result.stuID%TYPE,
p_curID IN t_result.curID %TYPE
)
RETURN INTEGER
AS
    v_result INT;                -- 学生成绩
BEGIN
    /*查询指定学生编号和课程编号的学生成绩*/
    SELECT result INTO v_result
    FROM t_result
    WHERE stuID = p_stuID
    AND curID = p_curID;
    /*当查询到的学生成绩大于100 或者小于0时，返回错误信息*/
    IF v_result >100 OR v_result <0 THEN
        RETURN -1;
    ELSE
        /*返回课程成绩*/
        RETURN v_result ;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('stuID is not exsit!',SYSDATE, 'admin'););
    WHEN OTHERS THEN
```

## 零基础学SQL

```
DBMS_OUTPUT.PUT_LINE ('发生其他错误');
INSERT INTO t_log          -- 将错误信息写入日志表中
VALUES('Oracle error!' ,SYSDATE, 'admin'););
END get_result t;
```

这段PL/SQL语句是创建了一个参数模式为IN模式的函数get\_result。在该函数中，含有两个形式参数，p\_stuID和p\_curID分别表示学生编号和课程编号。函数声明中的RETURN子句返回的数据类型为VARCHAR2类型的。在关键字AS之后，声明了一个表示学生成绩的标量变量。

在BEGIN部分使用SELECT语句查询学生的课程成绩，如果查询的学生成绩大于100或者小于0，就返回-1，否则就将该学生的课程成绩返回给调用者。在EXCEPTION部分用来对抛出的异常进行捕获。在关键字END后面跟的是创建函数时的名字get\_result。

当这个函数被调用时，函数会接收学生编号和课程编号两个变量值，然后根据这两个变量的值在t\_result中对其成绩进行查询，并根据取得的v\_result值返回相应的结果。

### 23.1.4 创建有输出参数的函数

如果在创建函数时，函数中的参数模式指定为OUT模式，那么在完成函数的调用后，实际参数的值将返回给函数的调用者。在该函数中，被指定为OUT模式的参数的作用相当于一个变量，该形式参数必须被赋值。其值既可以被读取，也可以被写入。下面这个例子中，创建一个参数模式为OUT模式的函数，查询教师津贴。

例23.3 创建有输出参数的函数，查询教师津贴。

```
CREATE OR REPLACE FUNCTION get_pension
(p_teaID t_teacher. teaID%TYPE,          -- 教师编号
pension OUT REAL                          -- 教师津贴
)
RETURN VARCHAR2
AS
    v_teaName t_teacher .teaName%TYPE
BEGIN
    /* 查询指定教师编号的教师津贴*/
    SELECT teaName ,pension INTO v_teaName pension
    FROM t_teacher
    WHERE teaID= p_teaID;
    /*返回教师姓名 */
    RETURN v_teaName;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('teaID is not exist!' ,SYSDATE, 'admin'););
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!' ,SYSDATE, 'admin');
END get_pension;
```

这段PL/SQL语句是创建了一个参数模式为OUT模式的函数get\_pension。在该函数中，含有两个形

式参数，p\_teaID和pension分别表示教师编号和教师津贴。其中，p\_teaID参数模式为IN，pension的参数模式为OUT。函数声明中的RETURN子句返回的数据类型为VARCHAR2类型的。

在BEGIN部分使用SELECT语句查询指定教师编号的教师及其对应津贴信息，并将查询到的教师信息使用RETURN子句返回。在EXCEPTION部分对抛出的异常进行捕获。

当这个函数被调用时，其实际参数的值就会丢失。当完成函数的调用之后，形式参数的值就被赋值给实际参数。

**注意** 函数中如果参数的模式为OUT模式，则被定义为OUT模式的形式参数的数据类型是不能有NOT NULL约束的。如果将一个有NOT NULL约束的数据指定为OUT模式的形式参数，则会抛出VALUE\_ERROR异常。

### 23.1.5 创建有输入输出参数的函数

如果在创建函数时，函数中的参数模式指定为IN OUT模式，那么它会向函数传递初始值，同时也会向调用者返回更新后的结果值。即在调用该函数时，实际参数的值将传递给被调用的函数；在完成函数的调用后，实际参数的值将返回给函数的调用者。在该函数中，被指定为IN OUT模式的参数的作用相当于一个已初始化的变量，其值既可以被读取，也可以被写入。下面的例子中，创建一个参数模式为IN OUT模式的函数，查询更新后的教师工资。

**例23.4** 创建有输入输出参数的函数，查询更新后的教师工资。

```
CREATE OR REPLACE FUNCTION get_newSalary
(p_teaID t_teacher. teaID%TYPE,      -- 教师编号
p_c_rate IN OUT REAL                 -- 教师工资比率
)
RETURN VARCHAR2
AS
    v_salary t_teacher . salary %TYPE
BEGIN
    /* 更新教师工资*/
    UPDATE t_teacher
    SET salary = salary+salary* p_c_rate
    WHERE teaID= p_teaID;
    /*返回更新后的教师工资 */
    RETURN v_salary;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log              -- 将错误信息写入日志表中
        VALUES('teaID is not exsist!',SYSDATE, 'admin'););
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log              -- 将错误信息写入日志表中
        VALUES('Oracle error!',SYSDATE, 'admin');
END get_newSalary;
```

这段PL/SQL语句是创建了一个参数模式为IN OUT模式的函数get\_newSalary。在该函数中，含有两



## 零基础学SQL

个形式参数，p\_teaID和p\_c\_rate分别表示教师编号和教师工资比率。其中，p\_teaID参数模式为IN，p\_c\_rate的参数模式为IN OUT。函数声明中的RETURN子句返回的数据类型为VARCHAR2类型的。

在BEGIN部分使用UPDATE语句对指定教师的教师工资进行更新，并返回更新后的教师工资。在EXCEPTION部分对抛出的异常进行捕获。当这个函数被调用时，其实际参数的值就会丢失。当完成函数的调用之后，形式参数的值就被赋值给实际参数。

### 23.1.6 创建记录类型作为返回值的函数

在创建函数的PL/SQL语句中，也可以使用记录变量作为返回值。例如下面的这个例子中就是将查询到的学生信息通过记录返回。

**例23.5** 创建以记录变量作为输入参数的函数，返回查询到的学生信息。

```
CREATE OR REPLACE FUNCTION get_student
(p_stuID t_student.stuID %TYPE)
RETURN t_student %ROWTYPE
AS
    student_record t_student %ROWTYPE;
BEGIN
    /*查询学生信息*/
    SELECT stuID,stuName,age, sex birth INTO student_record
    FROM t_student
    WHERE stuID = p_stuID;
    /*返回学生记录*/
    RETURN student_record;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('stuID is not exist!',SYSDATE, 'admin'););
    WHEN OTHERS THEN              -- 捕获异常
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log          -- 将错误信息写入日志表中
        VALUES('Oracle error!',SYSDATE, 'admin');
END get_student;
```

这段PL/SQL语句创建了一个向学生表中插入数据记录的函数get\_student。在这个函数中，将学生编号作为输入参数传递给函数get\_student。并在声明的RETURN子句中指定返回的数据类型为一个记录类型。

在BEGIN部分，根据传入的参数p\_stuID的值查询学生信息，并将查询到的结果返回给记录student\_record。EXCEPTION部分用来捕获异常。在关键字END后面跟的是创建函数时的名字get\_student。

### 23.1.7 创建集合类型作为返回值的函数

集合变量也可以作为函数的返回值。例如下面的这个例子中就是将查询到的教师信息通过使用嵌套表返回。

**例23.6** 创建以集合变量作为返回值的函数，返回查询到的教师信息。



```
CREATE OR REPLACE FUNCTION get_teacherTable
(p_deptID t_teacher.deptID %TYPE)
RETURN teacherName_table_type
AS
    teacherName_table teacherName_table_type;
BEGIN
    /*查询教师信息*/
    SELECT teaID, teaName INTO teacherName_table
    FROM t_teacher
    WHERE deptID = p_deptID;
    /*返回教师姓名*/
    RETURN teacherName_table;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log
        VALUES('teaID is not exist!',SYSDATE, 'admin');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log
        VALUES('Oracle error!',SYSDATE, 'admin');
END get_teacherTable;
```

这段PL/SQL语句创建了一个查询到的教师信息的函数get\_teacherTable。在这个函数中，使用嵌套表作为函数get\_teacherTable的输入参数（这里省略了参数模式的IN关键字）。并在声明的RETURN子句中指定返回的数据类型为一个嵌套表类型。

在BEGIN部分，使用SELECT语句查询指定院系的教师姓名，将其放入到嵌套表teacherName\_table中，并使用RETURN子句返回嵌套表teacherName\_table中的教师信息。EXCEPTION部分用来捕获可能发生的异常。在关键字END后面跟的是创建函数时的名字get\_teacherTable。

当使用PL/SQL嵌套表作为数据表中数据列的数据类型时，首先需要使有CREATE TYPE命令创建一个嵌套表类型，而且还要在创建数据表时为以嵌套表作为数据类型的列指定一个存储表。所以在使用嵌套表作为函数返回值之前，还需要创建嵌套表类型。

```
CREATE TYPE teacher_table_type IS TABLE OF VARCHAR(15);
```

## 23.2 调用函数

函数的调用与存储过程的调用不同，函数调用只能是作为表达式的一部分被调用，而存储过程调用本身就是一个PL/SQL语句。而且函数还可以在SQL语句中调用。这一节将介绍在PL/SQL语句块中函数的调用方法。有关函数在SQL语句中的调用可以参看23.4.2小节。

### 23.2.1 调用无参数的函数

在PL/SQL中，如果该函数没有参数，可以直接通过函数名对其进行调用。例如，对于例23.1中显示当前系统日期和时间的函数，在PL/SQL语句块中就可以使用下面的方法对其进行调用。

```
BEGIN
```

```
DBMS_OUTPUT.PUT_LINE (sysdata_time);      -- 调用函数
END;
```

在这个PL/SQL语句块中只包含一个可执行部分BEGIN。其中，sysdata\_time就是函数的名字。最后以关键字END结束。其显示的结果如下所示。

```
2009-09-28 12:13:40
```

### 23.2.2 调用带有输入参数的函数

在一个函数中，如果该函数中带有输入参数，那么在调用该函数时，需要为函数中的输入参数提供数据值。例如，如果调用例23.2中返回学生的课程成绩的函数get\_result，可以使用下面的方法完成。

```
BEGIN
    DBMS_OUTPUT.PUT_LINE (get_result ('s102203', 't105'));      -- 调用函数
END;
```

在这个PL/SQL语句块中只包含一个可执行部分BEGIN。其中，get\_result是函数的名字。在该函数名之后的括号中包含有两个参数。值s102203表示学生编号，该值会赋值给函数中的第一个参数p\_stuID；值t105是课程编号，该值会赋值给函数中的第二个参数p\_curID。使用DBMS\_OUTPUT.PUT\_LINE将函数返回的结果显示输出。最后以关键字END结束。其显示结果如下所示。

```
85
```

调用get\_result函数后，显示输出的结果是85。表示学生编号为s102203的学生其课程号t105的课程成绩为85分。

PL/SQL中函数可以作为表达式的一部分而被调用。上面的例子中函数get\_result也可以像变量一样被使用。例如，下面的这个PL/SQL语句块。

```
BEGIN
    CASE
        WHEN get_result ('s102203', 't105')>90 THEN
            DBMS_OUTPUT.PUT_LINE ('A');
        WHEN get_result ('s102203', 't105')>80 THEN
            DBMS_OUTPUT.PUT_LINE ('B');
        WHEN get_result ('s102203', 't105')>70 THEN
            DBMS_OUTPUT.PUT_LINE ('C');
        WHEN get_result ('s102203', 't105')>60 THEN
            DBMS_OUTPUT.PUT_LINE ('D');
        ELSE
            DBMS_OUTPUT.PUT_LINE ('E');
        END CASE ;
END;
```

在这个PL/SQL语句块，是对函数get\_result中返回的值进行判断，其函数的返回值是由传入的参数决定。对于不同的成绩显示不同的评分等级。其显示的结果如下所示。

```
B
```

### 23.2.3 调用带有输出参数的函数

在一个函数中，如果该函数中带有输出参数，那么在完成对该函数的调用之后，函数中指定为输出参数的形参的值会赋值给实参。例如，如果调用例23.3中查询教师津贴的函数get\_pension，可以使用下面的方法完成。

```
DECLARE
    v_teaName t_teacher. teaName%TYPE;
    v_pension t_teacher. pension %TYPE;
BEGIN
    v_teaName:=get_pension ('t156354', v_pension);           -- 调用函数
    DBMS_OUTPUT.PUT_LINE ('教师姓名: ' || v_teaName);
    DBMS_OUTPUT.PUT_LINE ('津贴为: ' || v_pension);
END;
```

在这个PL/SQL语句块中，DECLARE部分首先声明了一个表示教师姓名的变量v\_teaName和一个表示教师津贴的变量v\_pension，在BEGIN部分调用了函数get\_pension，并为其中的输入参数p\_teaID赋值为't156354'，当函数调用结束之后，PL/SQL就会把修改后教师津贴的值赋给实参v\_pension。其显示结果如下所示。

```
教师姓名: 张昌
津贴为: 300
```

### 23.2.4 调用带有输入输出参数的函数

在一个函数中，如果该函数中带有输入输出参数，那么在调用该函数时，实参的值将传递给被调用的函数；在完成函数的调用后，实参的值将返回给函数的调用者。例如，对于调用例23.4中的查询更新后的教师工资的函数get\_newSalary，可以使用下面的方法完成。

```
DECLARE
    v_salary t_teacher. salary %TYPE;
    c_rate NUMBER;
BEGIN
    v_salary:= get_newSalary ('t156354', c_rate);           -- 调用函数
    DBMS_OUTPUT.PUT_LINE ('更新后的教师工资为: ' || v_salary);
    DBMS_OUTPUT.PUT_LINE ('更新的工资比率为: ' || c_rate);
END;
```

在这个PL/SQL语句块中，DECLARE部分首先声明了一个表示教师工资的变量v\_salary和一个表示教师工资增长比率的常量c\_rate，在BEGIN部分调用了函数get\_newSalary，并为其中的输入参数p\_teaID赋值为't156354'，当函数调用结束之后，PL/SQL就会把修改后教师工资比率的值赋给实参c\_rate。其显示结果如下所示。

```
更新后的教师工资为: 4180
更新的工资比率为: 0.1
```

### 23.2.5 调用返回值为记录类型的函数

在PL/SQL中，可以调用返回值为记录类型的函数，通过调用记录类型做返回值的函数，可以向其

## 零基础学SQL

调用的应用程序返回整行的数据记录。例如，对于调用例23.5中的返回查询到的学生信息的函数get\_student，可以使用下面的方法完成。

```
DECLARE
    student_record t_student %ROWTYPE;
BEGIN
    student_record := ge_student(get_student('s131203'));    -- 调用函数
    DBMS_OUTPUT.PUT_LINE ('学生编号为: ' || student_record.stuID);
    DBMS_OUTPUT.PUT_LINE ('学生姓名为: ' || student_record.stuName);
    DBMS_OUTPUT.PUT_LINE ('学生年龄为: ' || student_record.age);
    DBMS_OUTPUT.PUT_LINE ('学生性别为: ' || student_record.sex);
    DBMS_OUTPUT.PUT_LINE ('学生出生日期为: ' || student_record.birth);
END;
```

在这个PL/SQL语句块中，DECLARE部分声明了一个记录类型的变量。在BEGIN部分调用get\_student函数，并将该函数的返回值赋值给记录student\_record，最后将记录中的数据显示输出。其显示结果如下：

```
学生编号为: s131203
学生姓名为: 彭庄
学生年龄为: student_record.age = 23
学生性别为: student_record.sex = 男
学生出生日期为: student_record.birth = 19860212
```

### 23.2.6 调用返回值为集合类型的函数

在PL/SQL中，可以调用返回值为集合类型的函数，通过调用集合类型做返回值的函数，可以向其调用的应用程序返回多行的数据记录。例如，对于调用例23.6中的返回查询到的教师信息的函数get\_teacherTable，可以使用下面的方法完成。

```
DECLARE
    teacherName_table teacherName_table_type;
BEGIN
    teacherName_table := get_teacherTable('t_10');    -- 调用函数
    FOR i IN 1.. teacherName_table.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE ('教师编号为: ' || teacherName_table(i));
    END LOOP;
END;
```

在这个PL/SQL语句块中，DECLARE部分声明了一个嵌套表类型的变量。在BEGIN部分调用get\_teacherTable函数，并将该函数的返回值赋值给嵌套表teacherName\_table，最后将嵌套表teacherName\_table中的数据显示输出。其显示结果如下：

```
教师姓名: 张昌
教师姓名: 赵伟
教师姓名: 毛翠
教师姓名: 干波
```



## 23.3 参数传递

PL/SQL中，其参数传递的方式与存储过程中参数传递的方式相同。在函数的调用时，既可以使用参数名称也可以使用参数位置来为实参提供数据。通过参数位置或者参数名称可以把实参和形参关联起来。本节就来介绍有关函数参数传递的内容。

### 23.3.1 使用参数位置传递参数值

在PL/SQL中，可以使用参数位置传递参数变量或者数据。通过使用参数位置传递变量或者数据时，PL/SQL编译器会将第一个实参的值与第一个形参的值相关联，第二个实参的值与第二个形参的值相关联，按照参数定义的顺序依次为形参赋值。例如，对于例23.2中返回学生的课程成绩的这个函数get\_result，通过参数位置传递，可以使用下面的方法对其进行调用。

```
DECLARE
    v_stuID VARCHAR2(15) := 's281234';
    v_curID VARCHAR2 (15) := 't321';
    v_result NUMBER;
BEGIN
    v_result := get_result (v_stuID, v_curID);           -- 调用函数
END;
```

### 23.3.2 使用参数名称传递参数值

在PL/SQL中，可以使用参数名称传递参数变量或者数据。可以通过使用关联参照符=>将实参和形参关联起来。例如，对于例23.2中返回学生的课程成绩的函数get\_result，通过参数名称传递，可以使用下面的方法对其进行调用。

```
DECLARE
    v_stuID VARCHAR2(15) := 's281234';
    v_curID VARCHAR2 (15) := 't321';
    v_result NUMBER;
BEGIN
    v_result := get_result(p_stuID=> v_stuID, p_curID=> v_curID); -- 调用函数
END;
```

这个PL/SQL语句块中，通过使用参数名称为参数传递变量。在DECLARE部分定义了两个变量v\_stuID和v\_curID，分别用来表示学生编号和课程编号，并为其赋初值。在BEGIN部分，调用get\_result函数，并通过关联参照符=>为函数中的参数提供变量。

### 23.3.3 使用参数名称和位置传递参数值

在PL/SQL中，也可以混合使用参数名称和参数位置来传递参数变量或者数据。在混合使用参数名称和参数位置来传递参数变量或者数据时，参数位置传递参数值的方法需要在参数名称传递参数值的方法之前使用。例如，对于例23.2中返回学生的课程成绩的函数get\_result，混合使用参数名称和参数位置传递，可以使用下面的方法对其进行调用。

```
DECLARE
    v_stuID VARCHAR2(15) := 's281234';
```



## 零基础学SQL

```
v_curID VARCHAR2 (15) := 't321';
v_result NUMBER;
BEGIN
    v_result := get_result(p_stuID=> v_stuID, p_curID=> v_curID); -- 调用函数
END;
```

这个PL/SQL语句块中，调用函数get\_result时混合使用参数名称和参数位置传递两种传递方法。其中，第一个参数v\_curID是按参数位置传递，第二个参数是按参数名称传递。如果将第一个参数和第二个参数传递的方法反过来，那么在调用函数get\_result时就会出现错误。例如，下面的函数调用是不合法的。

```
DECLARE
    v_stuID VARCHAR2(15) := 's281234';
    v_curID VARCHAR2 (15) := 't321';
    v_result NUMBER;
BEGIN
    v_result := get_result(p_stuID=> v_stuID ,v_curID); -- 函数调用不合法
END;
```

### 23.4 函数在SQL中的应用

一般情况下，子程序（包括存储过程和函数）是不能在SQL语句中被调用的，但是如果一个函数满足某些条件的限制，那么该函数是可以在SQL语句中被调用的。这一节就来介绍在SQL语句中如何使用函数。

#### 23.4.1 纯度规则

如果希望函数可以在SQL语句中被调用，函数就必须满足一些不同的限制条件。在SQL语句中调用函数必须遵守以下的纯度规则。

- ☐ SQL语句中调用的函数必须是无参函数或者是带IN模式参数的函数。在SQL语句中不能调用都有OUT模式或者是IN OUT模式参数的函数。
- ☐ SQL语句中调用的函数中的形式参数不能使用PL/SQL中特有的数据类型（例如，BOOLEAN、RECORD、TABLE等），而只能使用SQL支持的标准数据类型（例如，NUMBER、CHAR、VARCHAR2、DATE、ROW等）。
- ☐ SQL语句中调用的函数的返回值类型也必须是SQL支持的标准数据类型（例如，NUMBER、CHAR、VARCHAR2、DATE、ROW等）。
- ☐ SQL语句中调用的函数不能包含SQL事务控制语句（例如，COMMIT、ROLLBACK等）、系统控制语句（例如，ALTER SYSTEM等）、会话控制语句（例如，SET ROLE等）以及数据定义语句（例如，CREATE等）。
- ☐ SQL语句中调用的函数必须是存储在数据库中的函数，该函数或者是独立存储，或者是作为包的一部分。
- ☐ SQL语句中调用的函数不能修改数据库中的任何数据表。
- ☐ 在INSERT、UPDATE或DELETE语句中调用函数时，不能对这些语句所能影响到的数据表进行查询或者修改操作。

在SQL语句中调用函数时，必须完全遵守上述的规则，如果在SQL语句中调用函数时，与上述规则发生冲突，就会在运行时得到一个错误。

### 23.4.2 在SQL中调用函数

在23.4.1节中，了解了在SQL语句中调用函数必须遵守的纯度规则。这一小节通过一个例子来看一下如何在SQL语句中调用函数。在SQL语句调用函数之前，首先需要创建一个函数，这里创建一个get\_curName函数，通过课程编号作为输入参数，查询该课程编号对应的课程名。

例23.7 创建函数，用来查询课程编号对应的课程名。

```
CREATE OR REPLACE FUNCTION get_curName
(p_curID IN t_curriculum.curID %TYPE
)
RETURN VARCHAR2
AS
    v_curName VARCHAR2 (10) ;           -- 课程名称
BEGIN
    /*查询指定课程编号对应的课程名称*/
    SELECT curName INTO v_curName
    FROM t_curriculum
    WHERE curID = p_curID;
    RETURN v_curName;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log              -- 将错误信息写入日志表中
        VALUES('stuID is not exsit!' ,SYSDATE, 'admin'););
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('发生其他错误');
        INSERT INTO t_log              -- 将错误信息写入日志表中
        VALUES('Oracle error!' ,SYSDATE, 'admin'););
END get_curName;
```

这段PL/SQL语句是创建了一个参数模式为IN模式的函数get\_curName。在该函数中，包含有一个形式参数p\_curID，用来表示课程编号。函数声明中的RETURN子句返回的数据类型为VARCHAR2类型的。在关键字AS之后，声明了一个表示课程名称的标量变量。

在BEGIN部分使用SELECT语句查询课程编号对应的课程名称，并将查询到的课程名称通过RETURN子句返回。在EXCEPTION部分用来对抛出的异常进行捕获。在关键字END后面跟的是创建函数时的名字get\_curName。

这个函数满足上面的在SQL语句中调用函数必须遵守的纯度规则，因此可以在SQL语句中对该函数进行调用。

```
SELECT curID,get_curName(curID) AS curName
FROM t_curriculum
```

这里在SELECT语句中调用了函数get\_curName，其查询的结果如下所示。

```
curID    curName
-----
```

t105	计算机系统结构
t232	数据库基础
t321	C语言
t333	高等数学
t542	操作系统

## 23.5 删除函数

与在SQL语句中删除一个数据表类似，函数也可以删除。使用DROP命令可以将函数从数据字典中删除。删除函数的语法格式如下：

```
DROP function_name ;
```

其中，function\_name为要删除函数的名字，这个函数一定是一个已经存在的函数。例如，要删除函数get\_result，就可以使用下面的语句来完成。

```
DROP get_result;
```

在执行该DROP命令时，程序会隐式执行一个COMMIT命令。如果函数不存在，则在使用DROP命令时会引发“Object does not exist”

## 23.6 存储过程与函数

PL/SQL中的子程序包括存储过程和函数。这里用两章的篇幅分别介绍了存储过程和函数的使用。它们在许多地方都有相同的特性。

- ☐ 存储过程和函数的程序代码都是由SQL语句或者过程语句组成的。
- ☐ 在其过程体和函数体中都可以包含声明部分、可执行部分和异常处理部分。
- ☐ 存储过程和函数都可以由应用程序进行调用。
- ☐ 存储过程和函数都可以通过使用参数位置、参数名称或者参数名称和位置组合的方式进行参数传递。
- ☐ 存储过程和函数都可以使用默认值或NOCOPY编译提示的方式传递参数。（有关使用默认值或是NOCOPY编译提示的方式传递参数的方法可以参考22.4.4小节和22.4.5小节，函数使用默认值或是NOCOPY编译提示的方式传递参数与存储过程的相同）。

虽然存储过程和函数在许多地方都有相同的特性，但是它们之间在一些方面还是存在不同的。存储过程和函数的不同主要表现在以下几个方面。

- ☐ 存储过程调用本身就是一个PL/SQL语句，作为一个独立的部分来执行；而函数调用只能是作为表达式的一部分被调用，它既可以在PL/SQL语句块中被调用，也可以在SQL语句中调用。
- ☐ 存储过程的RETURN子句没有返回值，在存储过程中的RETURN子句的作用只是用来表示程序执行是否成功，它并不会返回任何的值或者表达式。而函数体中的RETURN子句需要返回一个特定的数据值。
- ☐ 如果返回的值有多个，可以通过OUT参数，使用存储过程来完成；如果返回值只有一个，可以通过函数体中的RETURN子句，使用函数来完成。

**说明** 函数也可以通过OUT参数返回多个值，但是在实际应用中，并不建议使用这种方式。如果要返回多个值，则建议使用存储过程，通过指定OUT参数来完成。（可以参看例22.2.2小节以及22.3.3小节）

## 23.7 小结

本章主要介绍了函数的创建、调用、参数传递以及在SQL语句中的应用等内容。在本章中主要讨论了函数的创建、删除以及调用的方法，在创建函数时，在其声明部分需要使用一个RETURN子句指定函数返回的数据类型，在函数体内还需要至少一个包含表达式RETURN子句。还介绍了通过使用参数位置、参数名称以及参数名称和位置组合的方式传递参数的方法。

与存储过程不同，函数除了可以在PL/SQL语句块中被调用之外，也可以在SQL语句中被调用。因此本章的23.4节中专门介绍了函数在SQL语句中的应用。包括纯度规则以及如何在SQL语句中调用函数。最后对存储过程和函数的异同进行了比较。



## 第24章 包

在PL/SQL中，包是一个模式对象，通过一定的逻辑组合，可以将相关的类型、常量、变量、存储过程、函数和异常组合在一起。一个包由两部分组成：包说明和包体。这两个部分是相互独立的。每一个部分在数据字典中都可以单独存储。PL/SQL中的包具有模块化、可以隐藏实施细节、方便应用程序设计、重载子程序以及良好的性能等特点。这一章就来介绍PL/SQL中的包。

本章重点：

- ☐ 包的创建
- ☐ 包中公有元素的调用
- ☐ 包中子程序的重载
- ☐ 包的删除方法
- ☐ Oracle数据库中常用内置系统包

### 24.1 创建包

PL/SQL中的包由包说明和包体两个独立的部分组成。包说明部分是应用程序的接口，主要包含包的内容信息，在包说明中声明了变量、常量、游标、自定义数据类型、存储过程或者函数；包体部分主要用来实现包说明中的内容，而且在包体部分还可以定义自己私有的游标、存储过程或者函数。包体不是必需的。本节就来介绍包说明和包体的创建。

#### 24.1.1 创建包说明

在PL/SQL中，如果要创建一个包，首先需要创建包说明，然后再创建包体，这一小节就来介绍包说明的创建。创建包说明可以使用CREATE [OR REPLACE] PACKAGE语句来完成。创建包说明的语法规则如下：

```
CREATE [OR REPLACE] PACKAGE package_name
{IS | AS}
    [variable_declaration ...]           -- 声明变量
    [cursor_declaration...]             -- 声明游标
    [exception_declaration ...]         -- 声明异常
    [object_declaration ...]            -- 声明对象
    [function_declaration...]           -- 声明函数
    [procedure_declaration...]          -- 声明过程
END [package_name];
```

其中，package\_name表示包的名称；关键字IS或者是AS之后用来声明变量、常量、游标、自定义数据类型、存储过程或者函数等这些元素。这些元素不必都存在。例如，在包说明中可以只有函数和



过程的声明，而不需要声明其他类型或者异常。

**注意** 在包说明中如果要声明函数或者是存储过程，那么该函数或者存储过程只能包含函数说明、过程说明和形式参数，不能包括函数体或者是过程体的代码。函数体或者是过程体的代码应该在包体中实现。

在包说明中声明的变量、常量、游标、自定义数据类型、存储过程或者函数等这些元素都是公有的，对应用程序来说，它们都是可见并且是可以调用的。因此，可以在不重新编译调用应用程序的前提下，对包体的实现进行修改。下面来看一个创建包说明的例子。

例24.1 创建包说明student\_package。

```
CREATE OR REPLACE PACKAGE student_package
AS
  g_curID VARCHAR2 (15);           -- 声明变量
  g_c_rate NUMBER:= 1.5;           -- 声明常量
  TYPE studentResult_array t_ result.result %TYPE
  INDEX BY BINARY_INTEGER;         -- 声明可变数组
  PROCEDURE ins_ result             -- 声明过程
  (p_stuID IN t_ t_result.stuID%TYPE,
  p_curID IN t_result.curID %TYPE
  );
  PROCEDURE delete_ result          -- 声明过程
  (p_stuID IN t_ t_result.stuID%TYPE,
  p_curID IN t_result.curID %TYPE
  );
  CREATE OR REPLACE FUNCTION get_student -- 声明函数
  (p_stuID t_student. stuID %TYPE)
  RETURN t_student %ROWTYPE;
  e_ illegalValue EXCEPTION;       -- 声明异常
  e_ illegalResult                  -- 声明异常
END student_package;
```

在包说明student\_package中包含了一个公有变量g\_curID、一个共有常量g\_c\_rate、一个可变数组studentResult\_array、两个公有的存储过程ins\_result和delete\_result、一个函数get\_student和两个异常e\_illegalValue、e\_illegalResult。在包说明student\_package中可以看到，在声明存储过程和函数时，这里只是包含了函数说明和过程说明及其对应的参数，并没有包含函数和存储过程的实现部分。

如果需要把某些内容声明为公有的，那么这些内容就需要放到包说明中。例如，在包说明student\_package中的常量g\_c\_rate就可以被其他的应用程序或者是PL/SQL语句块使用。

在创建包时，包说明一定要在包体之前定义，在编写包说明时，为了使其在以后的程序中可以多次使用，应该保证包的通用性。另外，由于在包说明中声明的变量、常量、游标、自定义数据类型、存储过程或者函数等这些元素都是公有的，因此对开发人员或者用户而言必须可见的部分才应该放在包说明中。

### 24.1.2 创建包体

包体部分主要用来实现包说明中的内容，在包体中包含了包说明中声明的每一个游标、函数或者

## 零基础学SQL

是存储过程的实现。在包体中也可以有自己的变量、常量、游标、自定义数据类型、存储过程或者函数，但是这些元素都是私有的，不能由其他的PL/SQL语句块或者是应用程序进行引用。可以使用CREATE PACKAGE BODY创建一个包体。创建包体的语法规则如下：

```
CREATE [OR REPLACE] PACKAGE BODY package_name
{IS | AS}
    -- 公有变量、常量、游标、自定义数据类型、存储过程或者函数的实现
    -- 定义私有的变量、常量、游标、自定义数据类型、存储过程或者函数
END package_name;
```

其中，package\_name表示包的名称，该名称需要与包说明中的名称相同；关键字IS或者是AS之后用来实现共有的变量、常量、游标、自定义数据类型、存储过程或者函数。也可以在包体中定义私有的变量、常量、游标、自定义数据类型、存储过程或者函数，它们不需要在包说明部分编写对它们的声明，对于包体来说它们都是本地的，这样的子程序只能在包内可见。在包外是不能被访问的。因此，在包体内实现的内容是不可见也是不可调用的。

**注意** 包体和包说明是相互分离的数据字典对象。在程序运行时，包说明必须要首先编译成功，包体才能通过编译。如果包说明不能编译成功，那么包体也不能被成功编译。

只有在包说明中声明之后，由包体实现的内容才能在包外被引用。可以在不重新编译调用应用程序的前提下，对包体的实现进行修改。下面这个例子中创建的就是student\_package的包体。

### 例24.2 创建student\_package的包体。

```
CREATE OR REPLACE PACKAGE BODY student_package
/*将学生的成绩信息插入到新表的存储过程*/
PROCEDURE ins_result
(p_stuID IN t_t_result.stuID%TYPE,
p_curID IN t_result.curID %TYPE
)
AS
    v_result INTEGER;                --学生成绩
BEGIN
    /*查询指定学生编号和课程编号的学生成绩*/
    SELECT result INTO v_result
    FROM t_result
    WHERE stuID = p_stuID
    AND curID = p_curID;
    /*当查询到的学生成绩大于100 或者小于0时，抛出e_illegalValue异常*/
    IF v_result >100 OR v_result <0 THEN
        RAISE e_illegalValue;
    ELSE
        /*将课程成绩插入到新的成绩表中*/
        INSERT INTO t_newResult
        VALUES(p_stuID,p_curID, v_result);
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
```

```
INSERT INTO t_log                                -- 将错误信息写入日志表中
VALUES('stuID is not exist!',SYSDATE,'admin');
WHEN e_illegalValue THEN
DBMS_OUTPUT.PUT_LINE ('查询成绩不合法 (大于100或者小于0)');
INSERT INTO t_log                                -- 将错误信息写入日志表中
VALUES('result is illegal!',SYSDATE,'admin');
END ins_result;
/*删除学生课程成绩的存储过程*/
PROCEDURE delete_result
(p_stuID IN t_t_result.stuID%TYPE,
p_curID IN t_result.curID %TYPE
)
AS
BEGIN
    /*删除指定学生编号和课程编号的学生成绩*/
    DELETE FROM t_result
    WHERE stuID = p_stuID
    AND curID = p_curID;
    /*如果该学生对应的课程成绩不存在*/
    IF SQL%NOTFOUND THEN
        RAISE e_illegalResult;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log                                -- 将错误信息写入日志表中
        VALUES('stuID is not exist!',SYSDATE,'admin');
    WHEN e_illegalResult THEN
        DBMS_OUTPUT.PUT_LINE ('查询成绩不合法 (大于100或者小于0)');
        INSERT INTO t_log                                -- 将错误信息写入日志表中
        VALUES('result is illegal!',SYSDATE,'admin');
END delete_result;
/*查询学生信息的函数*/
FUNCTION get_student
(p_stuID t_student.stuID %TYPE)
RETURN t_student %ROWTYPE
AS
    student_record t_student %ROWTYPE;
BEGIN
    /*查询学生信息*/
    SELECT stuID,stuName,age, sex birth INTO student_record
    FROM t_student
    WHERE stuID = p_stuID;
    /*返回学生记录*/
    RETURN student_record;
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
        INSERT INTO t_log                                -- 将错误信息写入日志表中
        VALUES('stuID is not exist!',SYSDATE,'admin');
```

## 零基础学SQL

```
END get_student;  
END student_package;
```

在这个student\_package包体中，实现了在包说明中声明的存储过程和函数。在包体中实现包说明中的存储过程或者函数时，包体中的存储过程或者函数的名称和参数列表（包括参数名和参数模式）要完全匹配，如果包体中实现的存储过程或者函数的名称和参数列表与包说明中的不匹配，程序运行时系统就会报错。

在创建包时，包体不是必需的。如果在一个包说明中只是声明了一些变量、常量、数据类型以及异常说明的内容，那么，包体部分就可以不需要了。不过，在包体中可以对包说明中声明的内容进行初始化。初始化部分在包体的声明部分之后，一般是用于初始化包中变量。例如，下面的这个例子创建的就是一个没有包体的包。

```
CREATE OR REPLACE PACKAGE no_body_package  
AS  
    TYPE stuentRec IS RECORD(  
        stuID          t_student.stuID%TYPE,  
        stuName        t_student.stuName%TYPE,  
        age            t_student.age%TYPE,  
        sex            t_student.sex%TYPE,  
        birth          t_student.birth%TYPE  
    );  
    curID t_result curID%TYPE;  
    result INTEGER;  
    e_illegalResult    EXCEPTION;  
    e_illegalstudentID EXCEPTION;  
END no_body_package;
```

在这个包说明中，只是声明了一个记录类型、两个变量和两个异常说明，它们都没有相应的实现部分，所以包体也就没有必要创建了。一般情况下，创建这样的包说明是为其他的应用程序或者触发器提供全局变量（有关触发器的内容可以参看第25章）。

能够直接在包体内定义子程序而不用在包说明部分编写它们的说明。但是，这样的子程序只能在包内使用。

## 24.2 调用包中的公有元素

在前面的一节中已经介绍过，在包说明中声明的变量、常量、游标、自定义数据类型、存储过程或者函数都是公有的。既然这些元素是公有的，那么就可以被其他的应用程序所调用。例如，可以在PL/SQL语句块中调用student\_package包的ins\_result存储过程。调用方法如下：

```
BEGIN  
    student_package.ins_result('s102203','t105');  
END;
```

从调用语句中，可以看到在PL/SQL中调用包中存储过程与调用独立的存储过程的区别。在调用包中的存储过程时，需要在存储过程的前面加上一个包名称的前缀。

在PL/SQL中也可以调用包中公有的函数和公有的变量。其调用方法与调用包中公有的存储过程的



方法相同。要想调用student\_package包的get\_student函数，就可以使用下面的PL/SQL语句块来完成。

```
DECLARE
    stu_record t_student%ROWTYPE;
BEGIN
    stu_record := student_package.get_student ('s102203');
    DBMS_OUTPUT.PUT_LINE ('学生编号: ' || stu_record.stuID);
    DBMS_OUTPUT.PUT_LINE ('学生姓名: ' || stu_record.stuName);
    DBMS_OUTPUT.PUT_LINE ('学生年龄: ' || stu_record.age);
    DBMS_OUTPUT.PUT_LINE ('学生性别: ' || stu_record.sex);
    DBMS_OUTPUT.PUT_LINE ('出生日期: ' || stu_record.birth);
END;
```

函数get\_student返回的是一个记录类型。在这个PL/SQL语句块中，DECLARE部分首先需要声明一个stu\_record记录类型的变量，在BEGIN部分调用了student\_package包中的get\_student函数，并将函数返回的信息显示输出。其显示的结果如下：

```
学生编号: 's102203'
学生姓名: 赵亮
学生年龄: 23
学生性别: 男
出生日期: 1986-05-16
```

如果要想调用student\_package包的公有变量g\_curID，并为其赋值，就可以使用下面的PL/SQL语句块来完成。

```
BEGIN
    student_package.g_curID:= 't232';
END;
```

在调用包中的公有变量时，同样也需要在其前面加上包名作为前缀。调用包中的这些公有的存储过程或者函数，就可以实现所需要的功能。开发人员或者用户不需要对包中存储过程或者函数的实现细节进行考虑。

### 24.3 在包中使用重载

在PL/SQL中，存储过程和函数可以在包中重载。包的重载特性允许在一个包内定义多个名称相同但是参数不同的子程序（包括存储过程和函数）。也就是说，不同的子程序可以有相同的名字，只要保证其形参的参数数量、参数顺序或者是数据类型不同即可（类似于Java语言的方法重载）。当调用包中的某一个子程序时，PL/SQL会比较形参列表和实参列表，并根据形参列表和实参列表的比较结果选择合适的子程序。

例如，现在希望查询这一个月某门课程需要多少学时，可以通过课程编号来查询，也可以通过课程名称和该课程所占的学分数来查询。这就可以使用重载来完成。

```
CREATE OR REPLACE PACKAGE learntime_package          -- 包说明
AS
PROCEDURE get_courseLearntime
(p_curID IN t_curriculum.curID %TYPE
);
```



## 零基础学SQL

```
PROCEDURE get_courseLearntime
(p_curName IN t_curriculum. curName %TYPE,
p_credit IN t_curriculum. credit %TYPE
);
END learntime_package;
/
CREATE OR REPLACE PACKAGE BODY learntime_package -- 包体
/*通过课程编号来查询课程学时*/
PROCEDURE get_courseLearntime
(p_curID IN t_curriculum.curID %TYPE
)
AS
v_learntime t_curriculum%TYPE;
BEGIN
SELECT learntime INTO v_learntime
FROM t_curriculum;
WHERE curID = p_curID;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
INSERT INTO t_log -- 将错误信息写入日志表中
VALUES('curID is not exsist!' ,SYSDATE,'admin');
END;
/*通过课程名称和该课程所占的学分来查询课程学时*/
PROCEDURE get_courseLearntime
(p_curName IN t_curriculum. curName %TYPE,
p_credit IN t_curriculum. credit %TYPE
)
AS
v_learntime t_curriculum%TYPE;
BEGIN
SELECT learntime INTO v_learntime
FROM t_curriculum;
WHERE curName = p_curName
AND credit = p_credit;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ('查询的数据不存在');
INSERT INTO t_log -- 将错误信息写入日志表中
VALUES('curName is not exsist!' ,SYSDATE,'admin');
END;
END learntime_package;
```

在这个包learntime\_package中，有两个同名的存储过程get\_courseLearntime，第一个存储过程接收的参数p\_curID，在数据表中是一个VARCHAR2类型的参数。第二个存储过程接收两个参数，第一个参数p\_curName，在数据表中是VARCHAR2类型；第二个参数p\_credit，在参数表中是INTEGER类型。每一个存储过程根据不同的参数都可以用来查询课程的学时信息。

例如，如果知道某一门课程的课程编号，希望通过课程编号来查询某一门课程的学时数，就可以

通过下面的PL/SQL语句块来完成。

```
BEGIN
    learntime_package.get_courseLearntime('t105');
END;
```

如果不知道课程编号，但是知道课程名称和该课程所占的学分，也可以使用learntime\_package包中的另一个存储过程来查询。

```
BEGIN
    learntime_package.get_courseLearntime('计算机系统结构',4);
END;
```

上面的例子中，是在包中重载了两个存储过程，当然也可以在包中对函数进行重载。从上面的例子中可以看到，通过使用重载，使得同样的操作可以运用于不同类型的对象，方便了应用程序的开发。

如果要在包中使用重载特性，必须保证名称相同的子程序中的参数数量、参数顺序或者是数据类型不同。在上面的例子中，两个名为get\_courseLearntime存储过程的参数个数是不相同的。如果参数个数是相同的，但是参数的类型不同也是可以的。例如，下面的存储过程的重载也是合法的。

```
FUNCTION get_salary(id NUMBER);
FUNCTION get_salary(name VARCHAR2);
```

但是，在包中对子程序使用重载时，还是有一些限制的。在下面的这些情况下，使用重载将是不合法的。

- ☐ 如果两个子程序参数类型相同，只是参数的名称和指定的参数方式不同，则不能进行重载。例如，下面的这两个存储过程的重载就是非法的。

```
PROCEDURE get_courseLearntime(p_curName VARCHAR2)
PROCEDURE get_courseLearntime(p_curID VARCHAR2)
```

- ☐ 两个子程序虽然参数名称不相同，但是其参数类型都是同一个类型的（例如，都是字符类型、数字类型等），也不能进行重载。例如，下面的这两个存储过程的重载就是非法的。

```
PROCEDURE get_courseLearntime(p_curName CHAR)
PROCEDURE get_courseLearntime(p_curID VARCHAR2)
```

这里CHAR和VARCHAR2都属于字符类型。

- ☐ 如果两个函数只是返回值类型不同，则不能进行重载。例如，下面的这两个函数的重载就是非法的。

```
FUNCTION get_salary RETURN NUMBER;
FUNCTION get_salary RETURN VARCHAR2;
```

## 24.4 删除包

使用DROP命令也可以删除包。如果只是想删除包体，而不想删除包说明，可以使用DROP PACKAGE BODY命令。删除包体的语法规则如下：

```
DROP PACKAGE BODY packagebody_name ;
```

其中，packagebody\_name为要删除的包体的名字。例如，要删除包体student\_package，就可以使

## 零基础学SQL

用下面的语句来完成。

```
DROP PACKAGE BODY student_package;
```

如果想将包说明及其包体一并删除，可以使用DROP PACKAGE命令。删除包说明及其包体的语法格式如下：

```
DROP PACKAGE package_name ;
```

其中，package\_name为要删除的包的名字，这个包一定是一个已经存在的包。例如，要删除student\_package包，就可以使用下面的语句来完成。

```
DROP PACKAGE student_package;
```

在执行该DROP命令时，程序会隐式执行一个COMMIT命令。如果存储过程不存在，则在使用DROP命令时会引发“Object does not exist”。

## 24.5 系统包

在Oracle数据库及其Oracle工具中包括许多内置的系统包，例如，前面的章节中经常出现的DBMS\_OUTPUT就是一个用于提供输入输出信息的包。通过这些内置的系统包可以为PL/SQL的应用程序实现许多其他的功能。例如，文件读写、会话通信等功能。这一章就来简要介绍一些其中比较重要的包。

### 24.5.1 生成并发送报警信息的包DBMS\_ALERT

DBMS\_ALERT包是用于生成并发送报警信息。通过使用DBMS\_ALERT包可以在特定的数据库值发生变化时，将报警信息传递给应用程序。

**注意** 只有SYS和SYSTEM权限的用户以及被授予EXECUTE权限的其他用户才可以使用DBMS\_ALERT包。

对于没有SYS和SYSTEM权限的用户要想使用DBMS\_ALERT包，需要被授予EXECUTE权限。可以使用GRANT命令为其他用户授予EXECUTE权限。例如，为用户‘admin’1授予EXECUTE权限，就可以使用下面的语句来完成。

```
GRANT EXECUTE ON DBMS_ALERT TO 'admin'1;
```

下面对DBMS\_ALERT包中所包含的主要过程进行介绍。

#### 1. SIGNAL过程

可以通过DBMS\_ALERT包中的SIGNAL过程发送报警信息。该过程只有在事务执行提交操作时才会发送报警信息，在事务执行回退操作时不会发送报警信息。使用DBMS\_ALERT包中的SIGNAL过程的语法规则如下：

```
DBMS_ALERT.SIGNAL(name IN VARCHAR2,message IN VARCHAR2);
```

其中，参数name表示指定的报警事件的名称；参数message表示发送的报警信息。该信息的最大字节数为1800个字节。这两个参数的参数模式都是IN模式。下面的这个例子演示了SIGNAL过程的使用方法。

```
DECLARE
    v_alertName VARCHAR2:='Alert';
BEGIN
    DBMS_ALERT.SIGNAL(v_alertName,'This is the alert!');
    COMMIT;
END;
```

## 2. WAITANY和WAITONE过程

可以使用WAITANY或者是WAITONE过程对报警信息进行读取。其中，WAITANY表示该过程等待当前会话中任何一个报警事件发生，并且当该事件发生时输出相应的信息；WAITONE表示该过程可等待当前会话中某个特定报警事件发生，并且当该事件发生时输出相应的信息。WAITANY过程和WAITONE过程语法规则如下：

```
/* WAITANY过程*/
DBMS_ALERT.WAITANY(
    name OUT VARCHAR2,
    message OUT VARCHAR2,
    status OUT VARCHAR2,
    timeout IN NUMBER DEFAULT maxwait);
/* WAITONE 过程*/
DBMS_ALERT.WAITONE (
    message OUT VARCHAR2,
    message OUT VARCHAR2,
    status OUT VARCHAR2,
    timeout IN NUMBER DEFAULT maxwait);
```

其中，参数name表示指定的报警事件的名称；参数message表示发送的报警信息；参数status表示返回状态值。如果返回值为0，则表示发生了预警事件；如果返回值为1，则表示发生超时；参数timeout用来设置等待预警事件的超时时间。默认值为最大的超时时间86400000秒（1000天）。在这4个参数中，前三个参数的参数模式都是OUT，最后一个参数timeout的参数模式为IN。

**说明** 在执行WAITANY或者是WAITONE过程之前，程序会隐式地发出COMMIT命令。

## 3. REGISTER过程

在DBMS\_ALERT包中还提供了注册和删除预警事件的过程。通过REGISTER过程可以注册一个预警事件。其语法规则如下：

```
DBMS_ALERT.REGISTER(name IN VARCHAR2);
```

其中，参数name表示指定的报警事件的名称。

## 4. REMOVE和REMOVEALL过程

删除预警事件可以使用REMOVE或者REMOVEALL过程。其中，REMOVE过程用于删除一个会话中不需要的预警事件；REMOVEALL是删除当前会话中所有的预警事件。REMOVE过程和REMOVEALL过程的语法规则如下：

```
DBMS_ALERT.REMOVE(name IN VARCHAR2);
```



```
DBMS_ALERT. REMOVEALL;
```

### 5. SET\_DEFAULTS过程

通过DBMS\_ALERT包中的SET\_DEFAULTS过程可以设置检测预警事件的时间间隔。其语法规则如下：

```
DBMS_ALERT.SET_DEFAULTS (sensitivity IN NUMBER);
```

其中，sensitivity表示检测预警事件的时间间隔。其默认时间间隔为5秒。

## 24.5.2 输入和输出信息的包DBMS\_OUTPUT

包DBMS\_OUTPUT用来输入和输出信息，方便程序的测试与调试。其中，过程PUT和PUT\_LINES可以将信息发送到SGA的一个缓存中，过程GET\_LINE和GET\_LINES可以显示缓存中的信息。下面对DBMS\_OUTPUT包中所包含的主要过程进行介绍。

### 1. DISABLE和ENABLE过程

DBMS\_OUTPUT包中的DISABLE过程用于禁用过程PUT、PUT\_LINE、GET\_LINE和GET\_LINES。其语法规则如下：

```
DBMS_OUTPUT. DISABLE;
```

与DBMS\_OUTPUT包中的DISABLE过程相对应的是ENABLE过程。该过程用于激活对PUT、PUT\_LINE、GET\_LINE和GET\_LINES过程的调用。其语法规则如下：

```
DBMS_OUTPUT.ENABLE (buffer_size IN INT DEFAULT 20000);
```

其中，ENABLE过程中的参数表示调用该过程时缓存中默认的大小为20000。当调用该过程时，缓存的最大值可以为1000000，最小值为2000。

### 2. PUT和PUT\_LINE过程

DBMS\_OUTPUT包中的PUT过程用于分块建立行信息。使用PUT过程显示信息时，所有的信息将显示在同一行。其语法规则如下：

```
DBMS_OUTPUT. PUT(item IN NUMBER);  
DBMS_OUTPUT. PUT(item IN VARCHAR2);  
DBMS_OUTPUT. PUT(item IN DATE);
```

在DBMS\_OUTPUT包中重载了过程PUT。其中，参数item用来指定要显示输出的内容。在这三个重载的PUT过程中，都接收一个输入模式的参数。第一个PUT过程接收一个NUMBER类型的参数；第二个PUT过程接收一个VARCHAR2类型的参数；第三个PUT过程接收一个DATE类型的参数。

**说明** 当使用过程PUT显示信息时，如果想结束当前行，必须调用NEW\_LINE过程。NEW\_LINE过程用来在行的尾部追加一个行结束符。

DBMS\_OUTPUT包中的PUT\_LINE过程用于将一个完整行的信息写入到缓存中。使用PUT\_LINE过程显示信息时，当信息显示后，会自动在行尾加上一个行结束符，自动换行。其语法规则如下：

```
DBMS_OUTPUT. PUT_LINE (item IN NUMBER);
```



```
DBMS_OUTPUT.PUT_LINE (item IN VARCHAR2);
DBMS_OUTPUT.PUT_LINE (item IN DATE);
```

在DBMS\_OUTPUT包中也重载了过程PUT\_LINE。PUT\_LINE过程中的参数与PUT过程中的参数相同。

下面通过一个例子来看一下这两个过程在显示时的区别。

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('中国1949-2009 国庆60周年');
    DBMS_OUTPUT.PUT('中国1949-2009, ');
    DBMS_OUTPUT.PUT('国庆60周年');
    DBMS_OUTPUT.NEW_LINE;
END;
```

在这个PL/SQL语句块中分别使用了DBMS\_OUTPUT包中的PUT和PUT\_LINE过程显示信息。其显示的结果如下所示。

```
中国1949-2009 国庆60周年
中国1949-2009 , 国庆60周年
```

### 3. GET\_LINE和GET\_LINES过程

DBMS\_OUTPUT包中的GET\_LINE过程用来取得缓存中的单行信息。其语法规则如下：

```
DBMS_OUTPUT.GET_LINE(line OUT VARCHAR2,status OUT INTEGER);
```

其中，参数line表示存放在缓存中的单行信息，其最大字节数为255个字节；参数status用来确定过程是否执行成功。如果执行成功则返回0，否则返回1。这两个参数的参数模式都是输出模式。

DBMS\_OUTPUT包中的GET\_LINES过程用来取得缓存中的多行信息。其语法规则如下：

```
DBMS_OUTPUT.GET_LINES(lines OUT CHARARR,numlines INOUT INTEGER);
```

其中，参数lines表示存放在缓存中的多行信息，该参数模式是输出模式；参数numlines用于指定并存放要检索的行数。该参数的参数模式是输入输出模式。

### 24.5.3 进行管道通信的包DBMS\_PIPE

DBMS\_PIPE包用于在同一例程的不同会话之间进行管道通信，所谓管道可以认为是一块内存区域。不同的会话之间可以通过管道发送消息。两个会话可以不在同一台机器上，只要它们可以向服务器发送PL/SQL命令，那么这两个会话之间就可以进行通信。

**注意** 管道是异步的，一旦向管道发送了一条消息，就不能取消它。

如果用户要执行DBMS\_PIPE包中的过程和函数，则必须要为用户授予EXECUTE权限。可以使用GRANT命令为用户授予EXECUTE权限。例如，为用户'admin'1授予EXECUTE权限，就可以使用下面的语句来完成。

```
GRANT EXECUTE ON DBMS_PIPE TO 'admin'1;
```

下面对DBMS\_PIPE包中所包含的主要过程和函数进行介绍。

## 零基础学SQL

### 1. CREATE\_PIPE函数

DBMS\_PIPE包中的CREATE\_PIPE函数可以用于创建一个管道。该管道既可以是公有的，也可以是私有的。最终创建的管道是公有的还是私有的，取决于其参数private的值。CREATE\_PIPE函数的语法规则如下：

```
DBMS_PIPE.CREATE_TYPE(  
    pipename IN VARCHAR2,  
    maxpipesize IN INTEGER DEFAULT 8192,-  
    private IN BOOLEAN DEFAULT TRUE)  
RETURN INTEGER;
```

其中，参数pipename表示管道的名称；参数maxpipesize表示管道消息的最大尺寸，默认值为8192；参数private用来指定创建的管道是公有的还是私有的。如果private的值为TRUE，则表示创建的管道是私有的；如果private值为FALSE，则表示创建的管道是公有的。函数的返回值为INTEGER类型。如果返回的值为0，则表示创建管道成功；否则表示创建管道失败。例如，要创建一个名为myPipe的私有管道，就可以使用下面的语句完成。

```
DECLARE  
    v_flag INTEGER;  
BEGIN  
    v_flag :=DBMS_PIPE.CREATE_TYPE('myPipe',8192,TRUE);  
    IF v_flag=0 THEN  
        DBMS_OUTPUT.PUT_LINE('创建私有管道成功');  
    END IF;  
END;
```

### 2. PACK\_MESSAGE过程

为了能够将消息写入到管道中，首先需要将消息写入到本地消息缓冲区。可以使用DBMS\_PIPE包中的PACK\_MESSAGE过程将消息写入到本地消息缓冲区。其语法规则如下：

```
DBMS_PIPE.PACK_MESSAGE(item IN VARCHAR2);  
DBMS_PIPE.PACK_MESSAGE(item IN NCHAR);  
DBMS_PIPE.PACK_MESSAGE(item IN NUMBER);  
DBMS_PIPE.PACK_MESSAGE(item IN DATE);  
DBMS_PIPE.PACK_MESSAGE_RAW(item IN RAW);  
DBMS_PIPE.PACK_MESSAGE_ROWID(item IN ROWID);
```

在DBMS\_PIPE包中重载了过程PACK\_MESSAGE。其中，参数item表示指定写入的管道信息。其参数的模式为输入模式。第一个PACK\_MESSAGE过程接收一个VARCHAR2类型的参数；第二个PACK\_MESSAGE过程接收一个NCHAR类型的参数；第三个PACK\_MESSAGE过程接收一个NUMBER类型的参数；第四个PACK\_MESSAGE过程接收一个DATE类型的参数。另外两个过程分别为PACK\_MESSAGE\_RAW和PACK\_MESSAGE\_ROWID，PACK\_MESSAGE\_RAW过程接收一个ROW类型的参数；PACK\_MESSAGE\_ROWID过程接收一个RAWID类型的参数。

PACK\_MESSAGE过程支持在缓冲区中写入不同的数据类型。在PACK\_MESSAGE过程中既可以将字符类型、数字类型的消息写入到本地消息缓冲区中，也可以将日期类型的消息写入到本地消息缓冲区中。

### 3. SEND\_MESSAGE函数

在使用DBMS\_PIPE包中的PACK\_MESSAGE过程将消息写入到本地消息缓冲区之后，就可以使用SEND\_MESSAGE函数将本地消息缓冲区的内容发送到管道中。其语法规则如下：

```
DBMS_PIPE.SEND_MESSAGE(  
    pipename IN VARCHAR2,  
    timeout IN INTEGER DEFAULT MAXWAIT,-  
    maxpipesize IN INTEGER DEFAULT 8192)  
RETURN INTEGER;
```

其中，参数pipename表示管道的名称；参数timeout表示发送消息的超时时间，默认值为最大的超时时间86 400 000秒（1000天）；参数maxpipesize表示表示管道消息的最大尺寸，默认值为8192。函数的返回值为INTEGER类型。如果返回的值为0，则表示发送成功；如果返回值为1，则表示消息发送超时；如果返回值为3，则表示发送消息时出现中断。

### 4. RECEIVE\_MESSAGE函数

在管道的另一个端，通过RECEIVE\_MESSAGE函数可以接收管道消息，并将消息写入本地消息缓冲区中。其语法规则如下：

```
DBMS_PIPE.RECEIVE_MESSAGE(  
    pipename IN VARCHAR2,  
    timeout IN INTEGER DEFAULT MAXWAIT)  
RETURN INTEGER;
```

其中，参数pipename表示管道的名称；参数timeout表示发送消息的超时时间，默认值为最大的超时时间86 400 000秒（1000天）。如果timeout的值为0，则RECEIVE\_MESSAGE函数会立即返回。函数的返回值为INTEGER类型。如果返回的值为0，则表示发送成功；如果返回值为1，则表示消息发送超时；如果返回值为2，则表示本地缓冲区不能容纳管道消息；如果返回值为3，则表示发送消息时出现中断。

### 5. UNPACK\_MESSAGE过程

使用函数RECEIVE\_MESSAGE接收管道消息之后，使用UNPACK\_MESSAGE过程将消息缓冲区的内容写入到变量中，并读取缓冲区的消息。其语法规则如下：

```
DBMS_PIPE.UNPACK_MESSAGE(item OUT VARCHAR2);  
DBMS_PIPE.UNPACK_MESSAGE(item OUT NCHAR);  
DBMS_PIPE.UNPACK_MESSAGE(item OUT NUMBER);  
DBMS_PIPE.UNPACK_MESSAGE(item OUT DATE);  
DBMS_PIPE.UNPACK_MESSAGE_RAW(item OUT RAW);  
DBMS_PIPE.UNPACK_MESSAGE_ROWID(item OUT ROWID);
```

在DBMS\_PIPE包中也重载了过程UNPACK\_MESSAGE。其中，参数item表示接收管道信息的变量。其参数的模式为输出模式。

通过使用上面介绍的DBMS\_PIPE包中的存储过程和函数，就可以实现两个会话之间的管道通信了。首先通过PACK\_MESSAGE过程将消息写入到本地消息缓冲区，并使用SEND\_MESSAGE函数将本地消息缓冲区的内容发送到管道中。

```
DECLARE
```

## 零基础学SQL

```
v_pipeName VARCHAR2(20):='pipeTest';
v_status INTEGER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE('This is the pipe message');
    DBMS_PIPE.PACK_MESSAGE(1234);
    DBMS_PIPE.PACK_MESSAGE(SYSDATE);
    v_status := SEND_MESSAGE(v_pipeName);
    /*如果没有发送失败*/
    IF v_status !=0 THEN
        DBMS_OUTPUT.PUT_LINE('ERROR!' || v_status);
    END IF;
END;
```

在这个PL/SQL语句块中，使用PACK\_MESSAGE过程写入消息，这里写入了一个字符串消息、一个数字消息和一个日期消息，并通过SEND\_MESSAGE函数将这个由字符串、数字和日期组成的消息通过pipeTest的管道发送出去。如果SEND\_MESSAGE函数的返回值不为0，则表示发送消息失败，就显示消息发送失败的原因。如果SEND\_MESSAGE函数的返回值为0，则消息会成功发送到管道的另外一端。

在连接同一个数据库的另一个会话中，通过RECEIVE\_MESSAGE函数和UNPACK\_MESSAGE过程就可以接收来自pipeTest管道的消息。

```
DECLARE
    v_pipeName VARCHAR2(20):='pipeTest';
    v_status INTEGER;
    v_string VARCHAR2;
    v_number NUMBER;
    v_date DATE;
BEGIN
    v_status := RECEIVE_MESSAGE(v_pipeName);
    /*如果接收数据成功*/
    IF v_status =0 THEN
        DBMS_PIPE.UNPACK_MESSAGE(v_string);
        DBMS_PIPE.UNPACK_MESSAGE(v_number);
        DBMS_PIPE.UNPACK_MESSAGE(v_date);
        DBMS_OUTPUT.PUT_LINE('v_string:' || v_string);
        DBMS_OUTPUT.PUT_LINE('v_number:' || v_number);
        DBMS_OUTPUT.PUT_LINE('v_date:' || v_date);
    ELSEIF
        /*如果接收数据失败*/
        DBMS_OUTPUT.PUT_LINE('ERROR!' || v_status);
    END IF;
END;
```

在这个PL/SQL语句块中，使用RECEIVE\_MESSAGE函数接收管道消息，并将消息写入本地消息缓冲区中。如果函数的返回值为0，表示接收消息成功，就通过UNPACK\_MESSAGE过程将消息缓冲区的内容写入到变量中，并读取缓冲区的消息。如果函数的返回值不为0，表示接收消息失败，并显示接收消息失败的原因。如果接收消息成功，会显示如下的结果。

```
v_string: This is the pipe message
v_number:1234
```

v\_date:2009-09-30 17:48:32

## 6. NEXT\_ITEM\_TYP函数

NEXT\_ITEM\_TYP函数用于确定本地消息缓冲区下一项的数据类型。一般在调用RECEIVE\_MESSAGE函数之后调用该函数。其语法规则如下：

```
DBMS_PIPE.NEXT_ITEM_TYPE RETURN INTEGER;
```

该函数没有参数。其返回值类型为INTEGER。如果返回值为0，则表示管道中没有任何消息；如果返回值为6，则表示下一项的数据类型为NUMBER类型；如果返回值为9，则表示下一项的数据类型为VARCHAR2类型；如果返回值为11，则表示下一项的数据类型为ROWID类型；如果返回值为12，则表示下一项的数据类型为DATE类型；如果返回值为23，则表示下一项的数据类型为ROW类型。

## 7. REMOVE\_PIPE函数

DBMS\_PIPE包中的REMOVE\_PIPE函数用于删除已经建立的管道。其语法规则如下：

```
DBMS_PIPE.REMOVE_PIPE(pipename IN VARCHAR2) RETURN INTEGER;
```

其中，参数pipename表示要删除的管道的名称。返回值为INTEGER类型。如果返回值为0，则表示成功删除管道。如果返回值不为0，表示删除管道失败。

## 8. PURGE过程

DBMS\_PIPE包中的PURGE过程用于清除管道中的内容。其语法规则如下：

```
DBMS_PIPE.PURGE(pipename IN VARCHAR2);
```

其中，参数pipename表示管道的名称。

## 24.5.4 安排和管理PL/SQL语句块的包DBMS\_JOB

DBMS\_JOB包用于安排和管理PL/SQL语句块，使Oracle数据库可以在指定的时间执行特定的任务。这些PL/SQL语句块会交由Oracle数据库的后台进程负责处理。下面对DBMS\_JOB包中所包含的主要过程和函数进行介绍。

### 1. SUBMIT过程

DBMS\_JOB包中的SUBMIT过程用来创建一个新的作业。使用SUBMIT过程创建一个作业，不仅需要指定该作业要执行的操作，还有指定作业的下次运行日期以及运行作业的时间间隔等内容。SUBMIT过程的语法规则如下：

```
DBMS_JOB.SUBMIT(  
    job OUT BINARY_INTEGER,  
    what IN VARCHAR2,  
    next_date IN DATE DEFAULT SYSDATE,  
    interval IN VARCHAR2 DEFAULT 'NULL',  
    no_parse IN BOOLEAN DEFAULT FALSE,  
    instance IN BINARY_INTEGER DEFAULT any_instance,  
    force IN DEFAULT FALSE);
```

其中，参数job表示指定作业编号，是该作业的唯一标示；参数what表示指定作业要执行的操作



## 零基础学SQL

（可以是存储过程或者是SQL语句）；参数next\_date用来指定作业的下次运行日期，其默认值为系统时间；参数interval用来指定运行作业的时间间隔，其默认值为NULL；参数no\_parse表示是否解析与作业相关的过程，如果值为TRUE，表示此PL/SQL代码在它第一次执行时应进行语法分析，如果值为FALSE表示该PL/SQL代码应立即进行语法分析，其默认值为FALSE；参数instance表示哪个例程可以运行作业；参数force表示是否强制运行与作业相关的例程，默认值为FALSE。这些参数中，只有第一个参数job的参数模式是输出模式，其他参数的参数模式都是输入模式。下面是一个创建新的作业的例子。

```
DECLARE
    var jobID NUMBER;
BEGIN
    DBMS_JOB.SUBMIT(: jobID, 'insertDate', SYSDATE, 'SYSDATE+60/86400');
    COMMIT;
END;
```

在这段PL/SQL语句中创建并提交一个新的作业。其中，参数jobID表示自动生成的JOB\_ID；参数insertDate表示需要执行的存储过程，该存储过程用来向数据表中插入时间；参数SYSDATE表示初次执行该作业的时间，这里设定为系统时间；参数SYSDATE+60/86400表示运行作业的时间间隔，这里设定的时间间隔为60秒。COMMIT语句用来提交作业。

在这个作业中，需要一个insertDate的存储过程，该存储过程用来向数据表中插入时间。其存储过程的PL/SQL语句如下：

```
CREATE OR REPLACE PROCEDURE insertDate
AS
BEGIN
    INSERT INTO dateTable VALUES(SYSDATE);
    COMMIT;
END insertDate;
```

### 2. RUN过程

DBMS\_JOB包中的RUN过程用于执行一个作业。该作业必须是一个已经存在的作业。其语法规则如下：

```
DBMS_JOB.RUN(job IN BINARY_INTEGER);
```

其中，参数job表示由SUBMIT过程提交时返回的JOB\_ID的值。例如，如果想执行一个已经创建好的作业，可以使用如下的语句完成。

```
BEGIN
    DBMS_JOB.RUN(: jobID);
END;
```

其中，参数jobID表示由SUBMIT过程提交时返回的JOB\_ID的值。

### 3. CHANGE过程

DBMS\_JOB包中的CHANGE过程用于改变与指定作业相关的设置信息。CHANGE过程的语法规则如下：

```
DBMS_JOB.CHANGE(
    job IN BINARY_INTEGER,
```

```
what IN VARCHAR2,  
next_date IN DATE,  
interval IN VARCHAR2  
instance IN BINARY_INTEGER DEFAULT NULL,  
force IN BOOLEAN DEFAULT FALSE);
```

其中，参数job表示指定作业编号，是该作业的唯一标示；参数what表示指定作业要执行的操作（可以是存储过程或者是SQL语句）；参数next\_date用来指定作业的下次运行日期；参数interval用来指定运行作业的时间间隔；参数instance表示哪个例程可以运行作业，默认值为NULL；参数force表示是否强制运行与作业相关的例程，默认值为FALSE。这些参数中，所有参数的参数模式都是输入模式。

例如，现在希望改变作业编号是jobID的作业信息，将运行作业的时间间隔由原来的60秒调整为30秒。

```
BEGIN  
    DBMS_JOB.CHANGE (: jobID,NULL,NULL, 'SYSDATE+30/86400');  
    COMMIT;  
END;
```

#### 4. REMOVE过程

DBMS\_JOB包中的REMOVE过程用于删除一个作业，该作业需要是一个已经存在于作业队列中计划运行的作业。REMOVE过程的语法规则如下：

```
DBMS_JOB.REMOVE (job in BINARY_INTEGER);
```

其中，参数job表示由SUBMIT过程提交时返回的JOB\_ID的值。如果想删除一个已经存在于作业队列中的计划运行作业，可以使用如下的语句完成。

```
BEGIN  
    DBMS_JOB.REMOVE(: jobID);  
END;
```

#### 注意

REMOVE过程删除的是一个已经存在于作业队列中计划运行的作业，对于已经运行的作业不能由REMOVE过程删除。

### 24.5.5 处理LOB数据类型数据的包DBMS\_LOB

DBMS\_LOB包主要是用来处理LOB数据类型（例如，BFILE、BLOB、CLOB和NCLOB）的数据。可以对这些数据进行比较、复制、获取LOB数据某一个特定部分内容等操作。下面对DBMS\_LOB包中所包含的主要过程和函数进行介绍。

#### 1. APPEND过程

DBMS\_LOB包中的APPEND过程用于将源LOB数据追加到目标LOB的尾部。该过程适用于BLOB和CLOB类型，但不适用于BFILE类型。其语法规则如下：

```
DBMS_LOB.APPEND(  
    dest_lob IN OUT NOCOPY BLOB,  
    source_lob IN BLOB);
```

其中，参数dest\_lob表示目标LOB，该参数的参数模式为输入输出类型；参数source\_lob表示源LOB，该参数的参数模式为输入类型。

为了使用APPEND过程将源LOB数据追加到目标LOB的尾部，这里首先需要创建一个用来存储LOB类型数据的表。

```
CREATE TABLE lobTable (  
    lobID INT NOT NULL,  
    lobText CLOB NOT NULL  
);
```

这里使用CREATE TABLE语句创建了一个lobTable表。在这个数据表中一共有两列。其中，lobID是主键，用来唯一标示一个lobText文本值；lobText是一个CLOB数据类型的文本。

创建完成lobTable表之后，就需要向表中添加数据，这里使用INSERT INTO语句向lobTable表中添加如下几条数据。

```
INSERT INTO lobTable (lobID, lobText) VALUES ('1','ABCD');  
INSERT INTO lobTable (lobID, lobText) VALUES ('2','CD');  
INSERT INTO lobTable (lobID, lobText) VALUES ('3','BCD');  
INSERT INTO lobTable (lobID, lobText) VALUES ('4','CD');
```

在完成上述准备之后，就可以使用DBMS\_LOB包中的APPEND过程将指定的源LOB数据追加到目标LOB的尾部。这里将lobID为1的lobText文本内容作为源LOB数据，将其追加到lobID为2的lobText文本的尾部。

```
CREATE OR REPLACE PROCEDURE lobAppend  
AS  
    dest_clob CLOB;                                --定义表示源LOB数据的变量  
    source_clob CLOB;                              --定义表示目标LOB数据的变量  
BEGIN  
    SELECT lobText INTO dest_clob                  --取得源LOB数据  
    FROM lobTable  
    WHERE lobID = 1 FOR UPDATE;  
    SELECT lobText INTO source_clob                --取得目标LOB数据  
    FROM lobTable  
    WHERE lobID = 2 FOR UPDATE;  
    DBMS_LOB.APPEND(dest_clob, source_clob); -- 将源LOB数据追加到目标LOB的尾部  
    COMMIT;  
END lobAppend;
```

在这个lobAppend存储过程中，首先定义了CLOB类型的变量dest\_clob和source\_clob，分别用来表示源LOB数据的变量和目标LOB数据的变量。在BEGIN部分，使用两个SELECT语句查询lobTable表，获取源LOB数据和目标LOB数据。在每一个查询语句之后需要使用FOR UPDATE子句锁定lob列。然后使用APPEND过程将取得的源LOB数据追加到目标LOB的尾部。最后使用COMMIT语句进行提交。此时，查看lobTable表，会看到相应的数据记录已经发生了变化。

## 2. COMPARE函数

DBMS\_LOB包中的COMPARE函数用于比较两个相同类型的不同LOB变量的内容。其语法规则如下：

```
DBMS_LOB.COMPARE (  
    lob1 IN OUT LOB,  
    lob2 IN LOB,  
    amout IN INTEGER DEFAULT 4294967295  
    offset1 INTEGER := 1,  
    offset2 INTEGER := 1)  
RETURN INTEGER;
```

其中，参数lob1和lob2表示要比较的两个LOB类型的变量。它们的数据类型可以是BFILE、BLOB、CLOB和NCLOB；参数amout表示要比较的字符或者是字节的个数；参数offset1用来指定第一个变量lob1要比较的起始位置；参数offset2用来指定第二个变量lob2要比较的起始位置。它们的默认值都为1。函数的返回值为INTEGER类型的数据。如果函数返回0，则表示比较的两个变量的结果是相同的；如果函数返回-1，则表示第一个变量比较的结果小于第二个变量比较的结果；如果函数返回1，则表示第一个变量比较的结果大于第二个变量比较的结果。

```
REATE OR REPLACE PROCEDURE lobCompare  
AS  
    clob1    CLOB;  
    clob2    CLOB;  
    clob3    CLOB;  
    v_count  INTEGER;  
BEGIN  
    SELECT lobText INTO clob1  
    FROM lobTable  
    WHERE lobID = 2 FOR UPDATE;           -- 取得lobID为2对应的lobText值  
    SELECT lobText INTO clob2  
    FROM lobTable  
    WHERE lobID = 3 FOR UPDATE;           --取得lobID为3对应的lobText值  
    SELECT lobText INTO clob3  
    FROM lobTable  
    WHERE lobID = 4 FOR UPDATE;           --取得lobID为4对应的lobText值  
    v_count := DBMS_LOB.COMPARE (clob1, clob2);    -- 比较clob1与clob2  
    IF v_count == 0 THEN  
        DBMS_OUTPUT.PUT('clob1和clob2相等');  
    ELSE  
        DBMS_OUTPUT.PUT('clob1和clob2不相等');  
    v_count := DBMS_LOB.COMPARE (clob1, clob3);    -- 比较clob1与clob3  
    IF v_count == 0 THEN  
        DBMS_OUTPUT.PUT('clob1和clob3相等');  
    ELSE  
        DBMS_OUTPUT.PUT('clob1和clob3不相等');  
    COMMIT;  
END lobCompare;
```

在这个存储过程中，是通过使用COMPARE函数比较两个LOB变量是否相同。在BEGIN部分通过SELECT语句分别取得lobID为2、3和4时，表lobTable对应的lobText值，然后通过COMPARE函数对其进行比较，并将比较后取得的值返回给变量v\_count。在IF语句中，根据变量v\_count的值进行判断，如果v\_count值为0，则表示两个lobText值是相同的；如果lobText值不为0，则表示两个lobText值是不相等的。



## 零基础学SQL

存储过程lobCompare创建完成之后，在SQL\*plus中，可以使用EXEC命令对其进行调用，其显示结果如下所示

```
SQL> EXEC lobCompare;
clob1和clob2不相等
clob1和clob3相等
```

### 3. GETLENGT函数

DBMS\_LOB包中的GETLENGT函数用于取得LOB数据的长度。其语法格式如下：

```
DBMS_LOB.GETLENGT(lob IN LOB) RETURN INTEGER;
```

其中，参数lob是一个LOB类型的变量，它可以是BLOB、CLOB和NCLOB类型。该函数返回的是LOB数据的长度，其值是一个INTEGER类型的数据。例如，现在想取得lobID为2对应的lobText数据的长度，可以通过下面这个存储过程来实现。

```
REATE OR REPLACE PROCEDURE lobLength
AS
    clob    CLOB;
    v_length INTEGER;
BEGIN
    SELECT lobText INTO clob1
    FROM lobTable
    WHERE lobID = 2 FOR UPDATE;
    v_length:= DBMS_LOB.GETLENGT(clob);
    DBMS_OUTPUT.PUT('clob的长度为: ' || v_length);
END lobLength;
```

### 4. COPY过程

DBMS\_LOB包中的COPY过程用于从指定位置开始将源LOB的内容复制到目标LOB中。其语法规则如下：

```
DBMS_LOB.COPY (
    dest_lob IN OUT NOCOPY LOB,
    source_lob IN LOB);
amount IN INTEGER
dest_offset INTEGER := 1,
source_offset INTEGER := 1)
```

其中，参数dest\_lob表示目标LOB，参数source\_lob表示源LOB。它们的数据类型可以是BLOB、CLOB和NCLOB；参数amount表示源LOB中指定要复制的字符或者是字节的个数；参数dest\_offset表示目标LOB的偏移量；参数source\_offset用来表示源LOB的偏移量。它们的默认值都为1。

下面的这个例子中，使用COPY过程将lobID为1对应的lobText中前两个字符的内容添加到lobID为2对应的lobText的尾部。

```
REATE OR REPLACE PROCEDURE lobCopy
AS
    dest_clob    CLOB;
    source_clob   CLOB;
```



```
dest_length INTEGER;
BEGIN
  SELECT lobText INTO source_clob
  FROM lobTable
  WHERE lobID = 1 FOR UPDATE;
  SELECT lobText INTO dest_clob
  FROM lobTable
  WHERE lobID = 2 FOR UPDATE;
  dest_length:= DBMS_LOB.GETLENGTH(dest_clob);
  DBMS_LOB.COPY (dest_clob,source_clob, 2, dest_length+1,1);
  COMMIT;
END lobCopy;
```

在存储过程中lobCopy中，首先定义两个CLOB类型的变量和一个INTEGER类型的变量，分别用来表示目标LOB、源LOB和目标LOB的长度。在BEGIN部分，通过SELECT语句取得目标LOB和源LOB对应的值，使用GETLENGTH函数取得目标LOG的长度，并通过DBMS\_LOB包中的COPY过程将源LOB中前两个字符的内容复制到目标LOB中。此时，查看lobTable表，会看到相应的数据记录已经发生了变化。

## 24.6 小结

本章主要介绍了PL/SQL中有关包的创建、删除、包中公有元素的调用，以及函数在包中的重载等内容。包在PL/SQL的应用中会经常地使用到。包主要分为两个部分：包说明和包体。将公有的内容放到包说明中供其他的PL/SQL语句调用，可以将不希望用户使用的内容放到包体中。通过使用包，可以隐藏实施细节、方便应用程序设计。包中的包说明和包体两个部分是独立存储在数据字典中的，其依赖性要比存储过程和函数的限制要少的多，也体现了包在性能上的优势。

另外，在包中还可以实现函数的重载，使用多个名称相同但是参数不同的子程序（包括存储过程和函数）可以在一个包中定义。

在本章的最后，还介绍了Oracle数据库中几个比较常用的内置的系统包。Oracle数据库中的系统内置包有很多，这里只是简要地介绍了其中几个。如果想要了解更多的Oracle数据库的系统包，可以查阅其他的相关文档。

## 第25章 触 发 器

触发器是在数据库中独立存储的对象，与存储过程和函数相类似，触发器也有自己的声明部分、可执行部分和异常处理部分，但是它不接收参数。触发器是一种特殊的存储过程。触发器不能直接调用，而是通过事件激发。可以激发触发器的事件包括DML语句、DDL语句的数据库或者系统事件。

触发器可以调用SQL语句、存储过程和函数。通过使用触发器可以提高数据库的数据安全性、增强应用程序健壮性，可以帮助开发人员更好地管理数据库实现复杂的逻辑功能，而且还能提高应用程序的开发效率。本章就来介绍PL/SQL中的触发器。

本章重点：

- ☐ 触发器类型及其用途
- ☐ 触发器限制
- ☐ DML触发器的创建
- ☐ DDL触发器的创建
- ☐ INSTEAD OF触发器的创建
- ☐ 系统事件触发器的创建
- ☐ 触发器的管理

### 25.1 触发器简介

触发器存储在数据库中，它是一种特殊的被隐式执行的存储过程。当有触发事件发生时，Oracle数据库就会激发触发器，并隐式地执行触发器相应的代码。本节就来介绍触发器的类型、用途及其使用触发器的限制。

#### 25.1.1 触发器的类型

触发器的类型可以分为数据操作语言触发器（DML触发器）、数据定义语言触发器（DDL触发器）、INSTEAD OF触发器、复合触发器以及事件触发器五种类型。下面就来简单介绍一下这五种触发器。

- ☐ 数据操作语言触发器（DML触发器）：数据操作语言触发器是基于DML操作的由DML语句激发的触发器。当在数据表中执行INSERT、UPDATE或者DELETE操作时，都会激发DML触发器。根据DML语句影响的行数，DML触发器可以分为语句触发器和行触发器。其中，语句触发器是无论指定的DML语句会影响多少行，都只会被激发一次的触发器，也称为表触发器。根据触发的时间不同，语句触发器又包括BEFORE触发器（在执行DML语句之前被触发的触发器）和AFTER触发器（在执行DML语句之后被触发的触发器）。行触发器是当执行DML语句时，每作用一行就会被触发一次的触发器。包括BEFORE触发器、AFTER触发器和行限制触发器三种类型。如果需要审计表数据的编号，可以使用行触发器。

- ❑ 数据定义语言触发器（DDL触发器）：当执行CREATE、ALTER、DROP语句来创建、修改或者删除数据库中的对象（例如，权限、角色、表、用户、视图等）等语句时，会激发DDL触发器。该触发器可以用来帮助控制或者监控DDL语句。
- ❑ INSTEAD OF触发器：当INSTEAD OF触发器被激发时，它不会执行INSERT、UPDATE和DELETE操作，它只会执行触发器本身的操作。INSTEAD OF触发器只适用于视图。INSTEAD OF触发器可以用来修改一个不可更新的视图。
- ❑ 复合触发器：当在数据表中执行INSERT、UPDATE或者DELETE操作时，会激发复合触发器。复合触发器可以同时担当语句触发器和行触发器的角色。复合触发器可以用来辅助管理像需要排序这样较大的触发器事件。它是Oracle 11g新增的一种触发器类型。
- ❑ 事件触发器：当产生DDL事件或者系统事件时，都会触发系统触发器。DDL事件包括执行CREATE、ALTER、DROP语句来创建、修改或者删除数据库中的对象（例如，权限、角色、表、用户、视图等）。系统事件包括服务器的开启和关闭、发生服务器错误等。通过系统触发器可以跟踪系统事件，并可以将跟踪的结果反馈给用户。

### 25.1.2 触发器的用途

触发器是在数据库中独立存储的对象，这一点类似于第24章介绍的包。当触发事件发生时，Oracle数据库就会隐式地执行触发器中相应的代码。触发器可以实现增强数据库的数据安全、实现数据和参数完整性以及数据库的数据审计等方面的功能。触发器的用途主要体现在以下几个方面。

- ❑ 增强数据库的数据安全：可以基于不同的时间或者是数据库中的数据限制用户的某些操作。例如，限制用户在休息时间修改数据库中的数据。
- ❑ 实现复杂的数据完整性：数据完整性要保证数据库中数据要满足一定的商业逻辑。在数据库的设计阶段，可以通过使用数据库约束来保证数据的完整性。但是，对于复杂的商业逻辑，使用数据库约束是无法完成的。例如，要实现员工在增长工资时，其新工资一次增长的幅度不能超过原来工资的50%。类似这样的问题，就需要通过触发器来实现。
- ❑ 实现参照完整性：数据库中的参照完整性保证了主从表之间，在对其中一个表的数据操作不会对与之关联的表造成不利的影响。当需要对主从表之间实现级联更新时，就需要使用触发器来完成。
- ❑ 实现数据审计：通过DML触发器可以审计数据表中记录的数据变化，跟踪用户对数据库的操作。记录数据表中记录的改变以及是哪一个用户对数据表中的记录做了修改，监视非法或者是可疑的数据库活动。

使用触发器会有一定的风险。当激发一个触发器时，可能会带来连锁触发（cascading trigger）的问题。所谓连锁触发，就是当一个触发器A调用一个SQL语句时，该SQL语句会激发另一个触发器B，而触发器B在调用另一个SQL语句时，该SQL语句会激发另一个触发器C，依次类推，就会造成连锁触发。Oracle数据库允许出现这样的连锁触发的现象，但是对连锁触发的数量有一定限制，连锁触发的数量不能超过32，如果超过了32，Oracle数据库就会抛出异常。

### 25.1.3 触发器的限制

一个触发器由触发器声明和触发器主体两个部分组成。其中，触发器声明包括触发器名称、触发事件或者语句、触发对象和触发限制；触发器的主体是PL/SQL语句块。只要是在PL/SQL语句块中合法

的语句在触发器体中也是合法的。但是在使用触发器时，还需要一些限制。这些限制主要包括以下几个方面。

- ❑ 不可以使用任何事务控制语句。控制事务语句包括COMMIT、ROLLBACK、SAVEPOINT、SETTRANSACTION。虽然使用这些语句的触发器在PL/SQL编译器中可以编译通过，但是该触发器在被激发时会得到一个错误。而且，由触发器调用的存储过程和函数也不能使用任何事务控制语句。
- ❑ 不能声明任何LONG或者是LONG RAW类型的变量。声明为LONG或者是LONG RAW数据类型的列，其行触发器不能使用:new或:old伪记录或者数据的行。但是在触发器体中可以使用LOB列。
- ❑ 触发器体的最大尺寸不能大于32 760个字节。在编写触发器代码时，应该让触发器体保持尽量的小。
- ❑ 触发器可以访问的表或者列也有一些限制。

## 25.2 创建DML触发器

DML触发器可用来控制DML语句。当在数据表中执行INSERT、UPDATE或者DELETE操作之前或者之后，都会激发DML触发器。DML触发器可以分为语句触发器和行触发器。无论指定的DML语句会影响多少行，语句触发器只会被激发一次；而对于行触发器来说，DML语句每作用一行，就会被触发一次。本节就来介绍DML触发器中语句触发器和行触发器的创建方法。

### 25.2.1 创建语句触发器

当用户在向数据表中执行数据的插入、修改或者删除操作时，都会激发语句触发器。在执行DML语句时，无论该DML语句会影响多少行数据记录，语句触发器都只会被激发一次。创建语句触发器的规则如下：

```
CREATE OR REPLACE TRIGGER trigger_name
{ BEFORE|AFTER } INSERT[ OR DELETE OR UPDATE]
ON table_name
[DECLARE
    declaration_statements; ]
BEGIN
    execution_statements;
END [trigger_name];
```

其中，trigger\_name表示触发器的名字；BEFORE或者AFTER子句表示该语句触发器是在执行DML语句之前还是在执行DML语句之后触发。BEFORE子句表示语句触发器是在执行DML语句之前触发，AFTER子句表示该语句触发器是在执行DML之后触发。INSERT、UPDATE和DELETE用来表示DML语句的触发事件，可以在多个事件之间使用OR关键字来创建一个语句触发器。当在多个事件之间使用OR关键字时，该触发器会参照多个事件来运行。table\_name用来表示执行DML语句所对应的表的名称。

在触发器体中，DECLARE语句用来声明变量、常量、游标、自定义数据类型等，它是可选的；在BEGIN部分中，可以包含赋值语句、流程控制语句以及与Oracle数据操作相关的语句。关键字END后面的trigger\_name表示触发器的名字，这个名字要与关键字TRIGGER后面的trigger\_name的名字一致。关键字END后面的trigger\_name是可选的。

**注意** 在语句触发器中不能使用WHEN子句，也不允许引用new或者old伪记录。否则会在编译时出现错误。

可以使用语句触发器对数据库的数据安全进行控制。例如，对于一个对数据表有更新权限的用户来说，他可以随时对该表执行更新操作。使用语句触发器可以限制该用户对数据表执行更新操作时间。下面的这个例子中就是限制用户在下班之后或者在周末对数据表进行更新操作。

**例25.1** 限制用户在下班之后或者在周末对dept表进行更新操作。

```
CREATE OR REPLACE TRIGGER trig_dept_editdata
BEFORE INSERT OR UPDATE OR DELETE
ON dept
DECLARE
    v_data NUMBER;
BEGIN
    SELECT TO_CHAR(SYSDATE, 'HH24') INTO v_data
    FROM dual;
    IF v_data > 17 OR v_data < 8 THEN
        RAISE_APPLICATION_ERROR(-20001, '不能在下班时间更新dept表');
    END IF;
    IF TO_CHAR(SYSDATE, 'DY', 'NLS_DATE_LANGUAGE=AMERICAN') IN ('SAT', 'SUN') THEN
        RAISE_APPLICATION_ERROR(-20002, '不能在周末时间更新dept表');
    END IF;
END trig_dept_editdata;
```

这里创建了一个名为trig\_dept\_editdata的触发器用于禁止用户在下班之后或者在周末更新dept表。在IF语句中使用了一个RAISE\_APPLICATION\_ERROR过程，它是DBMS\_STANDARD包中的一个过程，使用RAISE\_APPLICATION\_ERROR过程可以重新定义错误信息，并将应用程序中自定义的错误信息从服务器端传递给客户端。该过程可以直接使用，无须指定包名。其声明方法如下：

```
PROCEDURE RAISE_APPLICATION_ERROR
(error_number_in IN NUMBER, error_msg_in IN VARCHAR2);
```

RAISE\_APPLICATION\_ERROR过程需要两个参数，其中，参数error\_number\_in指定错误的代码，它是一个NUMBER类型的，其取值范围为-20 000~-20 999；参数error\_msg\_in用来显示自定义的错误信息，其错误信息的长度最多为2048字节。

在创建完trig\_result\_editdata触发器之后，为了验证该触发器，可以将系统的时间调整到18:00，然后执行下面的插入语句。

```
INSERT INTO dept
VALUES(90, '测试部', '大连')
```

这里向dept表中插入一条部分信息。由于时间已经超过了17:00，所以此时就会得到一个错误信息，提示不能对dept表执行插入操作。

```
ORA-20001: 不能在下班时间更新result表
ORA-06512: 在 "SCOTT. trig_dept_editdata", line 5
ORA-04088: 触发器 "SCOTT. trig_dept_editdata" 执行过程中出错
```

上面的例子中，是使用BEFORE子句表示语句触发器是在执行DML语句之前触发，也可以使用



AFTER子句表示该语句触发器是在执行DML之后触发。下面的这个例子中，就是通过AFTER来查询在对数据表进行更新操作之后，数据表中的数据记录的变化。

例25.2 查询在对数据表进行更新操作之后学生表中的数据记录。

```
CREATE OR REPLACE TRIGGER trig_student _count
AFTER INSERT OR UPDATE OR DELETE
ON t_student
DECLARE
    v_number NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_number
    FROM t_student;
    DBMS_OUTPUT.PUT_LINE ('学生表的记录总数为: ' || v_number);
END trig_student _count;
```

这里使用的是AFTER子句，表示在执行完DML语句之后，触发trig\_student \_count触发器，并将查询到的学生记录的结果显示出来。

### 25.2.2 创建行触发器

行触发器是当执行DML语句时，每作用一行就会被触发一次的触发器。由于DML语句不能记录列的变化，为了记录表数据的变化，就需要使用行触发器。但是行触发器的代码不能读取其对应基表中的数据。创建行触发器的语法规则如下：

```
CREATE OR REPLACE TRIGGER trigger_name
{ BEFORE|AFTER } INSERT、UPDATE和DELETE
ON table_name [REFERENCING OLD AS old|NEW AS new]
FOR EACH ROW
[WHEN (logical_expression)]
[DECLARE
    declaration_statements; ]
BEGIN
    execution_statements;
END [trigger_name];
```

其中，trigger\_name表示触发器的名字；BEFORE或者AFTER子句表示该行触发器是在执行DML语句之前还是在执行DML语句之后触发。BEFORE子句表示行触发器是在执行DML语句之前触发，AFTER子句表示该行触发器是在执行DML之后触发。INSERT、UPDATE和DELETE用来表示DML语句的触发事件，可以在多个事件之间使用OR关键字来创建一个行触发器。当在多个事件之间使用OR关键字时，该触发器会参照多个事件来运行。table\_name用来表示执行DML语句所对应的表的名称。

REFERENCING子句用来指定引用新、旧数据的方式。其中，old操作符表示引用旧数据，new操作符表示引用新数据。默认情况下，是使用old操作符。关键字FOR EACH ROW表示创建的是行触发器，WHEN子句用来限定触发条件，用来指定在什么条件下激发触发器。

在触发器体中，DECLARE语句用来声明变量、常量、游标、自定义数据类型等，它是可选的；在BEGIN部分中，可以包含赋值语句、流程控制语句以及与Oracle数据操作相关的语句。关键字END后面的trigger\_name表示触发器的名字，这个名字要与关键字TRIGGER后面的trigger\_name的名字一致。关键字END后面的trigger\_name是可选的。

从创建行触发器的语句中，可以看到其与创建语句触发器的不同。行触发器中比语句触发器多了一个FOR EACH ROW子句、一个WHEN子句以及old和new操作符。通过使用行触发器，可以获得数据表中每一行的new值和old值。可以通过该值来对数据进行审核。

在行触发器体中，通过:new和:old两个标识符，可以访问当前被处理行的数据。:new和:old是触发器体中的绑定变量。虽然编译器将它们当作记录类型，但是实际上它们并不是，因此通常把:new和:old称为伪记录。在INSERT、UPDATE和DELETE语句中，:new和:old伪记录在更新前后所表示的值是不同的。

- ❑ 当执行INSERT语句时，所有old伪记录的值都是NULL，即对于INSERT语句，:old伪记录是未定义的；:new伪记录表示INSERT语句完成之后被插入的值。
- ❑ 当执行UPDATE语句时，:old伪记录表示更新前的数据值；:new伪记录表示UPDATE语句完成之后被修改的值。
- ❑ 当执行DELETE语句时，:old伪记录表示行被删除的原始值；所有:new伪记录的值都是NULL，即对于DELETE语句，:new伪记录是未定义的。

行触发器可以实现主从表之间的级联更新操作，从而保证数据库的参照完整性。下面通过例子来看一下行触发器如何实现主从表的级联更新。

**例25.3** 实现主从表的级联更新，保证数据库的参照完整性。

```
CREATE OR REPLACE TRIGGER student_result_update_cascade
AFTER UPDATE OR DELETE OF stuID ON t_student
FOR EACH ROW
BEGIN
    UPDATE t_result
    SET stuID = :new.stuID
    WHERE stuID = :old.stuID;
END student_result_update_cascade;
```

这里创建一个student\_result\_update\_cascade触发器用于实现学生信息表和学生成绩表的级联更新操作。在这里使用了:old和:new两个标识符来取得更新前该数据行的原始值和更新语句完成之后，将要被更新的值。当在学生信息表t\_student中执行修改和删除操作的时候，都会触发该触发器。通过创建行触发器，实现主从表的级联更新操作，从而保证了数据库的参照完整性。

通过行触发器还可以对数据表中的数据进行审核，实现复杂的数据完整性约束。下面来看一个这方面的例子。

**例25.4** 审核数据表的变化，使得员工的新工资不得低于原有工资。

```
CREATE OR REPLACE TRIGGER trig_salary_update
AFTER UPDATE
ON emp
FOR EACH ROW
WHEN (:old.sal > :new.sal)
BEGIN
    RAISE_APPLICATION_ERROR(-20003, '员工的新工资不得低于原有工资');
END trig_salary_update;
```

在这个行触发器中，使用了WHEN子句的限定条件。在WHEN子句中，引用了new和old伪记录对员工的原有工资和新工资进行比较。当新工资低于原有工资时，就会触发RAISE\_APPLICATION\_

ERROR过程，显示相应的错误信息。

### 25.2.3 触发器谓词

在前面介绍的语句触发器和行触发器中，BEFORE子句或者是AFTER子句之后有时包含多个触发事件。例如在例25.3中就包含了UPDATE和DELETE两个触发事件，在例25.1和例25.2中，包含了INSERT、UPDATE和DELETE三个触发事件。为了区分具体是哪个事件触发了触发器，就需要使用触发器谓词。

触发器谓词包括INSERTING、UPDATING和DELETING三个，它们都是DBMS\_STANDARD包中的函数。这些函数可以直接使用，无须指定包名。这三个函数返回的都是一个布尔值，通过这三个布尔函数，可以用来区分到底是INSERT、UPDATE和DELETE三个触发事件中的哪一个触发了触发器。下面是对这三个触发器谓词的说明。

❑ INSERTING：当触发事件是INSERT操作时，返回值为TRUE，否则返回值为FALSE。

❑ UPDATING：当触发事件是UPDATE操作时，返回值为TRUE，否则返回值为FALSE。

❑ DELETING：当触发事件是DELETE操作时，返回值为TRUE，否则返回值为FALSE。

下面来看一个使用触发器谓词的例子。在这个例子中，根据不同的触发器谓词，向日志表中插入相关的数据更新信息。

例25.5 根据不同的触发器谓词，向日志表中插入相关的数据更新信息。

```
CREATE OR REPLACE TRIGGER trig_logInfo
AFTER INSERT OR UPDATE OR DELETE
ON dept
FOR EACH ROW
BEGIN
    CASE
        WHEN INSERTING THEN
            INSERT INTO t_log('对dept表执行插入操作',SYSDATE,user);
            DBMS_OUTPUT.PUT_LINE ('对dept表执行的操作为INSERT');
        WHEN UPDATING THEN
            INSERT INTO t_log('对dept表执行修改操作',SYSDATE,user);
            DBMS_OUTPUT.PUT_LINE ('对dept表执行的操作为UPDATE');
        WHEN DELETING THEN
            INSERT INTO t_log('对dept表执行删除操作',SYSDATE,user);
            DBMS_OUTPUT.PUT_LINE ('对dept表执行的操作为DELETE');
    END trig_logInfo;
```

在创建完trig\_logInfo触发器之后，可以在数据库中执行一条修改语句，这里执行下面的一条修改语句。

```
UPDATE dept SET dname = '测试与维护部'
WHERE deptno = 90;
```

其显示的结果为：

```
对dept表执行的操作为UPDATE
```

## 25.3 创建DDL触发器

当执行创建、修改或者删除数据库中的对象（例如，权限、角色、表、用户、视图等）等语句时，会激发DDL触发器。创建DDL触发器的语法规则如下：

```
CREATE OR REPLACE TRIGGER trigger_name
{ BEFORE|AFTER } DDL
ON table_name
[DECLARE
    declaration_statements; ]
BEGIN
    execution_statements;
END [trigger_name];
```

创建DDL触发器与创建DML触发器的语法基本相同。AFTER关键字后面的DDL表示支持DDL操作的语句。例如，CREATE TABLE、ALTER COMPIL、DROP TABLE。

通常，DDL触发器用于监控数据库中的重要事件。在项目开发和测试的过程中，用DDL触发器可以了解和监控数据库活动的动态。下面的这个例子中，就是创建一个用于防止用户删除自己对象的DDL触发器。

**例25.6** 防止用户删除自己模式中的对象。

```
CREATE OR REPLACE TRIGGER trig_drop
BEFORE DROP
ON mySchema.schema
BEGIN
    RAISE_APPLICATION_ERROR(-20004, '无效的删除操作');
END trig_drop;
```

这里创建的是在mySchema模式下的DDL触发器。该触发器用来防止用户删除自己模式中的对象。如果在创建完该触发器后，使用了下面的SQL语句。

```
DROP TABLE tableA;
```

其中，tableA是一个在tableA模式下创建的数据表。那么将得到如下的错误信息。

```
ORA-20004: 无效的删除操作
```

## 25.4 创建INSTEAD OF触发器

在视图中执行DML操作时需要一定的限制。简单视图中可以执行INSERT、UPDATE和DELETE操作，但是在基于分组函数、DISTINCT关键字、连接查询和集合查询的复杂视图中，不能执行INSERT、UPDATE和DELETE操作。为了在这些复杂视图中也可以执行INSERT、UPDATE和DELETE操作，就需要创建INSTEAD OF触发器。创建INSTEAD OF触发器的语法规则如下：

```
CREATE OR REPLACE TRIGGER trigger_name
INSTEAD OF INSERT[ OR DELETE OR UPDATE]
ON view_name
FOR EACH ROW
[WHEN (logical_expression)]
```

## 零基础学SQL

```
[DECLARE
    declaration_statements; ]
BEGIN
    execution_statements;
END [trigger_name];
```

在这个创建INSTEAD OF触发器的语句块中，关键字INSTEAD OF代替了DML触发器中的BEFORE和AFTER关键字；关键字ON后面跟的是视图的名字，而不再是table\_name表的名字。在创建INSTEAD OF触发器时，需要指定FOR EACH ROW 子句。

**注意** INSTEAD OF触发器只适用于视图。当INSTEAD OF触发器被激发时，不会执行INSERT、UPDATE和DELETE操作。

可以通过INSTEAD OF触发器编写更新视图的代码，来对一个不能执行INSERT、UPDATE和DELETE操作的视图执行更新操作。下面来看一个使用INSTEAD OF触发器的例子。

**例25.7** 创建INSTEAD OF触发器，可以对v\_teacher\_salary视图执行更新操作。

```
CREATE OR REPLACE TRIGGER trig_view_teacher_salary
INSTEAD OF INSERT
ON v_teacher_salary
FOR EACH ROW
DECLARE
    v_coun INT;
BEGIN
    SELECT COUNT(*) INTO v_count;
    FROM t_teacher
    WHERE teaID = :new.teaID;
    IF v_count = 0 THEN
        INSERT INTO t_teacher
        VALUES(:new.teaID,:new.teaName,:new.age,:new.sex,:new.deptID,
            :new.dept,:new.profession,new.salary,:new.pension);
    END IF;
END trig_view_teacher_salary;
```

这里创建了一个名为trig\_view\_teacher\_salary的INSTEAD OF触发器。在这个触发器中，如果插入的值在视图中不存在，就将其插入到视图中。

在这个INSTEAD OF触发器创建完成之后，就可以对该视图执行INSERT语句了。下面是向v\_teacher\_salary视图中插入一条记录的SQL语句。

```
INSERT INTO v_teacher_salary
VALUES('t103278','王以',30,'男','t_10','计算机系','讲师',3000,0);
```

执行完这条INSERT语句之后，可以通过SELECT语句查询到视图v\_teacher\_salary和数据表t\_teacher中数据记录的变化。

## 25.5 创建事件触发器

事件触发器是由系统事件或者是DDL事件激发的触发器。事件触发器包括系统事件触发器和用户



事件触发器。其中系统事件包括数据库的启动和登录、数据库的关闭和退出、服务器错误等，用户事件包括用户登录和注销、DDL事件等。这一节主要介绍事件触发器的创建。

### 25.5.1 事件属性函数

在创建事件触发器时，需要用到事件属性函数。事件属性函数在Oracle数据库的STANDARD包中声明和实现。在触发器中，可以像使用变量一样使用这些事件属性函数。下面就介绍一些常用的事件属性函数。

- ❑ `ORA_CLIENT_IP_ADDRESS`：返回客户端的IP地址。其返回的数据类型为VCHAR2类型。
- ❑ `ORA_DATABASE_NAME`：返回当前数据库名称。其返回的数据类型为VCHAR2类型。
- ❑ `ORA_DES_ENCRYPTED_PASSWORD`：返回DES加密后的用户口令。其数据类型为VCHAR2类型。
- ❑ `ORA_DICT_OBJ_NAME`：返回DDL操作对应的对象名称。
- ❑ `ORA_DICT_OBJ_NAME_LIST(name_list OUT ora_name_list_t)`：返回在特定事件中被修改的对象名的个数。该函数接受一个参数，其参数模式为输出模式，name\_list中包含触发事件所触发的对象名列表。其返回的数据类型为PLS\_INTEGER类型。
- ❑ `ORA_DICT_OBJ_OWNER`：返回DDL操作所对应对象的所有者名称。其返回的数据类型为VCHAR2类型。
- ❑ `ORA_DICT_OBJ_OWNER_LIST(owner_list OUT ora_name_list_t)`：返回在事件中被修改的对象所有者列表的元素个数。该函数接受一个参数，其参数模式为输出模式，owner\_list中包含触发器对象拥有者的列表。其返回的数据类型为PLS\_INTEGER类型。
- ❑ `ORA_DICT_OBJ_TYPE`：返回DDL操作所对应的数据库对象类型。其返回的数据类型为VCHAR2类型。
- ❑ `ORA_GRANTEE(user_list OUT ora_name_list_t)`：返回授权事件的授权者个数。该函数接受一个参数，其参数模式为输出模式，user\_list包含触发事件授予了权限或角色的用户列表。其返回的数据类型为PLS\_INTEGER类型。
- ❑ `ORA_INSTANCE_NUM`：返回当前数据库例程编号。其返回的数据类型为NUMBER类型。
- ❑ `ORA_IS_ALTER_COLUMN(column IN varchar2)`：检测特定列是否被修改。函数接受一个参数，其参数模式为输入模式，column表示一个列名。其返回的数据类型为BOOLEAN类型。如果列被修改，则返回TRUE，否则返回FALSE。
- ❑ `ORA_IS_CREATING_NESTED_TABLE`：检测是否正在建立嵌套表。其返回的数据类型为BOOLEAN类型。
- ❑ `ORA_IS_DROP_COLUMN(column IN varchar2)`：检测特定列是否被删除。函数接受一个参数，其参数模式为输入模式，column表示一个列名。其返回的数据类型为BOOLEAN类型。如果列被删除，则返回TRUE，否则返回FALSE。
- ❑ `ORA_IS_SERVERERROR(error_number)`：检测是否返回了特定Oracle错误。函数接受一个参数error\_number，该参数表示错误号。其返回的数据类型为BOOLEAN类型。
- ❑ `ORA_LOGIN_USER`：返回登录用户名。其返回的数据类型为VCHAR2类型。
- ❑ `ORA_PARTITION_POS`：该函数返回带SQL语句插入分区子句的数值位置，其返回类型为BINARY\_INTEGER。该函数只适用于INSTEAD OF CREATE触发器。

- ❑ **ORA\_PRIVILEGE\_LIST(privilege\_list OUT ora\_name\_list\_t)**：返回被授予或者被收回权限的用户个数。该函数接受一个参数，其参数模式为输出模式，privilege\_list包含触发事件授予的权限或角色的列表。其返回的数据类型为PLS\_INTEGER类型。
- ❑ **ORA\_REVOKE(user\_list OUT ora\_name\_list\_t)**：返回被收回权限的用户个数。该函数接受一个参数，其参数模式为输出模式，user\_list包含触发事件授予了权限或角色的用户列表。其返回的数据类型为PLS\_INTEGER类型。
- ❑ **ORA\_SERVER\_ERROR(position)**：返回错误堆栈中错误位置所对应的错误编号。接受一个参数position，该参数表示错误堆栈上的位置。其返回的数据类型为NUMBER类型。
- ❑ **ORA\_SERVER\_ERROR\_DEPTH**：返回错误堆栈中错误消息总数。其返回的数据类型为PLS\_INTEGER类型。
- ❑ **ORA\_SERVER\_ERROR\_MSG(position IN binary\_integer)**：返回错误堆栈中错误位置所对应的错误信息。接受一个参数position，该参数表示错误堆栈上的位置。其返回的数据类型为VARCHAR2类型。
- ❑ **ORA\_SERVER\_ERROR\_NUM\_PARAMS**：返回错误堆栈中错误位置被替换为错误消息的所有替代字符串个数。其返回的数据类型为PLS\_INTEGER类型。
- ❑ **ORA\_SERVER\_ERROR\_PARAM(position IN binary\_integer)**：返回错误堆栈中错误位置所对应的错误信息。接受一个参数position，该参数表示错误堆栈上的位置。其返回的数据类型为VARCHAR2类型。
- ❑ **ORA\_SQL\_TXT(txt\_list OUT ora\_name\_list\_t)**：返回触发器语句SQL文本的元素个数。该函数接受一个参数，其参数模式为输出模式，txt\_list包含触发事件的SQL语句处理的子串。其返回的数据类型为PLS\_INTEGER类型。
- ❑ **ORA\_SYSEVENT**：返回触发器的系统事件名称。其返回的数据类型为VARCHAR2类型。
- ❑ **ORA\_WITH\_GRANT\_OPTION**：检测授权是否带有WITH GRANT OPTION子句。其返回的数据类型为BOOLEAN类型，如果使用了WITH GRANT OPTION子句，则返回TRUE，否则返回FALSE。
- ❑ **SPACE\_ERROR\_INFO(error\_number OUT NUMBER, error\_type OUT VARCHAR2, object\_owner OUT VARCHAR2, table\_space\_name OUT VARCHAR2, object\_name OUT VARCHAR2, sub\_object\_name OUT VARCHAR2)**：检测错误是否与out-of-space条件相关。该函数接受6个参数，都为输出类型的参数。其返回类型为BOOLEAN类型。

以上是一些常用的事件属性函数及对函数用途的描述。下面通过几个例子来看一下如何在PL/SQL中使用这些事件属性函数。

#### 例25.8 取得客户端IP地址。

```
DECLARE
    v_address VARCHAR2(20);
BEGIN
    IF ORA_SYSEVENT = 'LOGON' THEN
        v_address := ORA_CLIENT_IP_ADDRESS;
    END IF;
END;
```

这段PL/SQL语句块是用来取得客户端的IP地址。在DECLARE部分声明了一个VARCHAR2类型的

变量v\_address。在BEGIN部分，使用了两个事件属性函数。其中，使用ORA\_SYSEVENT函数取得触发器的系统事件名称，如果触发器的系统事件名称为LOGON，就使用ORA\_CLIENT\_IP\_ADDRESS函数取得客户端IP地址，并将取得的客户端IP地址赋值给变量v\_address。

**例25.9** 取得被授予或者被收回权限的用户个数。

```
DECLARE
    v_counter PLS_INTEGER;
    v_privList DBMS_STANDARD.ORA_NAME_LIST_T;
BEGIN
    IF ORA_SYSEVENT = 'GRANT' OR ORA_SYSEVENT = 'REVOKE' THEN
        v_counter:= ORA_PRIVILEGE_LIST (v_privList);
    END IF;
END;
```

这段PL/SQL语句块是用来取得被授予或者被收回权限的用户个数。在DECLARE部分声明了两个变量，一个是PLS\_INTEGER类型的变量v\_counter，另一个是ORA\_NAME\_LIST\_T类型的变量v\_privList。因为在ORA\_PRIVILEGE\_LIST函数中需要一个形参数据类型为ORA\_NAME\_LIST\_T类型的变量。ORA\_NAME\_LIST\_T是一个VARCHAR2(64)数据类型的表。

在BEGIN部分，使用了两个事件属性函数。其中，使用ORA\_SYSEVENT函数用来取得触发器的系统事件名称，如果触发器的系统事件名称为GRANT或者是REVOKE，就使用ORA\_PRIVILEGE\_LIST函数取得被授予或者被收回权限的用户个数，并将取得的被授予或者被收回权限的用户个数赋值给变量v\_counter。

**例25.10** 取得错误堆栈中的错误编号和错误信息。

```
DECLARE
    v_errorMsg VARCHAR2(64);
    v_errorNum NUMBER;
BEGIN
    FOR i IN 1.. ORA_SERVER_ERROR_DEPTH LOOP
        v_errorMsg:= ORA_SERVER_ERROR_MSG (i);
        v_errorNum:= ORA_SERVER_ERROR (i);
    END LOOP;
END;
```

这段PL/SQL语句块是用来取得错误堆栈中的错误编号和错误信息。在DECLARE部分声明了两个变量，一个是VARCHAR2类型的变量v\_errorMsg，用来表示错误信息；另一个是NUMBER类型的变量v\_errorNum，用来表示错误编号。

在BEGIN部分，使用了三个事件属性函数。其中，ORA\_SERVER\_ERROR\_DEPTH函数用来取得错误堆栈中错误消息总数，并将取得的错误消息总数作为FOR循环的循环条件，取得所有的错误编号和错误信息。其中，使用ORA\_SERVER\_ERROR\_MSG函数取得错误信息文本信息，并将得到的错误信息文本信息赋值给变量v\_errorMsg；使用ORA\_SERVER\_ERROR函数取得错误编号，并将得到的错误编号赋值给变量v\_errorNum。

### 25.5.2 创建系统事件触发器

当发生数据库的启动和登录、数据库的关闭和退出、用户登录和注销等系统事件时，都会激发系

系统触发器。系统触发器可以很方便地跟踪每个用户和数据库服务器的运行时间。创建系统事件触发器的语法规则如下：

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} database_event ON {database | schema}
[DECLARE
    declaration_statements; ]
BEGIN
    execution_statements;
END [trigger_name];
```

在这个创建触发器的语句块中，关键字BEFORE和AFTER之后跟的是数据库系统事件。数据库系统事件主要包括STARTUP事件、SHUTDOWN事件、DB\_ROLE\_CHANGE事件和SERVERERROR事件。其中，STARTUP事件是在数据库启动后触发；SHUTDOWN事件是在数据库关闭之前触发；DB\_ROLE\_CHANGE事件是在改变角色后第一次打开数据库时触发；SERVERERROR事件是在发生Oracle错误时触发。

**注意** 系统事件触发器只能由sys用户创建。只有administrator database trigger系统权限的数据库管理员才可以创建系统事件触发器。

#### 例25.11 创建用于审计启动数据库事件的触发器。

```
CREATE OR REPLACE TRIGGER trig_startup_t_log
AFTER STARTUP ON DATABASE
BEGIN
    INSERT INTO t_log
    VALUES (ORA_SYSEVENT,SYSDATE, ORA_LOGIN_USER);
END trig_startup_t_log;
```

这段PL/SQL语句创建的是一个启动数据库时的事件触发器trig\_startup\_t\_log。其中，STARTUP表示启动数据库事件，在数据库启动后触发。数据库启动后，会向t\_log表中插入一条记录。在这条记录中，信息文本是ORA\_SYSEVENT函数返回的触发器的系统事件名称，时间为系统时间SYSDATE，用户名是ORA\_LOGIN\_USER函数返回的登录用户名。

#### 例25.12 创建用于审计关闭数据库事件的触发器。

```
CREATE OR REPLACE TRIGGER trig_shutdown_t_log
BEFORE SHUTDOWN ON DATABASE
BEGIN
    INSERT INTO t_log
    VALUES (ORA_SYSEVENT,SYSDATE, ORA_LOGIN_USER);
END trig_shutdown_t_log;
```

这段PL/SQL语句创建的是一个关闭数据库时的事件触发器trig\_shutdown\_t\_log。其中，SHUTDOWN表示关闭数据库事件，数据库关闭之前触发。在数据库关闭之前，会向t\_log表中插入一条记录。在这条记录中，信息文本是ORA\_SYSEVENT函数返回的触发器的系统事件名称，时间为系统时间SYSDATE，用户名为ORA\_LOGIN\_USER函数返回的登录用户名。

### 25.5.3 创建用户事件触发器

用户事件包括用户登录和注销、DDL事件等。用户事件触发器也只能由sys用户创建。其创建用户事件触发器的语法规则与创建系统事件触发器的语法规则相同。下面通过两个例子来看一下如何通过创建用户事件触发器审计用户登录和注销事件。

**例25.13** 创建用于审计用户登录事件的触发器。

```
CREATE OR REPLACE TRIGGER trig_login_ t_log
AFTER LOGIN ON DATABASE
BEGIN
    INSERT INTO t_log
    VALUES (ORA_CLIENT_IP_ADDRESS,SYSDATE, ORA_LOGIN_USER);
END trig_startup_ t_log;
```

这段PL/SQL语句创建的是一个用户触发器trig\_login\_t\_log。其中，LOGIN表示用户登录事件，用户登录事件后，会向t\_log表中插入一条记录。在这条记录中，信息文本是ORA\_CLIENT\_IP\_ADDRESS函数返回的客户端IP地址，时间为系统时间SYSDATE，用户名为ORA\_LOGIN\_USER函数返回的登录用户名。

**例25.14** 创建用于审计用户注销事件的触发器。

```
CREATE OR REPLACE TRIGGER trig_logoff_ t_log
BEFORE LOGOFF ON DATABASE
BEGIN
    INSERT INTO t_log
    VALUES (ORA_CLIENT_IP,SYSDATE, ORA_LOGIN_USER);
END trig_logoff_ t_log; SHUTDOWN
```

这段PL/SQL语句创建的是一个注销用户触发器trig\_logoff\_t\_log。其中，LOGOFF表示注销事件，在注销事件之前，会向t\_log表中插入一条记录。在这条记录中，信息文本是ORA\_CLIENT\_IP\_ADDRESS函数返回的客户端IP地址，时间为系统时间SYSDATE，用户名为ORA\_LOGIN\_USER函数返回的登录用户名。

## 25.6 管理触发器

前面的几节中，分别介绍了DML触发器、DDL触发器、INSTEAD OF触发器和事件触发器的创建，这一节来介绍触发器的管理和维护。管理和维护触发器主要包括触发器的禁用、触发器的激活、触发器的重新编译和删除。

### 25.6.1 禁用触发器

如果希望禁用触发器，可以使用ALTER TRIGGER...DISABLE命令。如果触发器被禁用，则在对数据表进行INSERT、UPDATE或者DELETE操作时，都不会执行触发器中的操作。禁用一个特定触发器的语法规则如下：

```
ALTER TRIGGER trigger_name DISABLE;
```

其中，trigger\_name表示触发器的名字。例如，如果想禁用trig\_dept\_editdata触发器，就可以使用





下面的语句完成。

```
ALTER TRIGGER trig_dept_editdata DISABLE;
```

上面的语句是禁用触发器trig\_dept\_editdata。如果希望禁用所有的触发器，可以使用ALTER TRIGGER...DISABLE ALL TRIGGERS命令。其语法规则如下：

```
ALTER TRIGGER trigger_name DISABLE ALL TRIGGERS;
```

### 25.6.2 激活触发器

如果希望将禁用的触发器重新激活，则可以使用ALTER TRIGGER...ENABLE命令。激活一个特定触发器的语法规则如下：

```
ALTER TRIGGER trigger_name ENABLE;
```

其中，trigger\_name表示触发器的名字。例如，如果想激活trig\_dept\_editdata触发器，就可以使用下面的语句完成。

```
ALTER TRIGGER trig_dept_editdata ENABLE;
```

上面的语句是激活触发器trig\_dept\_editdata。如果希望激活所有的触发器，可以使用ALTER TRIGGER...ENABLE ALL TRIGGERS命令。其语法规则如下：

```
ALTER TRIGGER trigger_name ENABLE ALL TRIGGERS;
```

### 25.6.3 重新编译触发器

当触发器为INVALID状态时（例如，使用ALTER TABLE语句修改表结构），该触发器是不能使用的。为了使用该触发器，就需要对其重新编译。重新编译一个特定触发器的语法规则如下：

```
ALTER TRIGGER trigger_name COMPILE;
```

其中，trigger\_name表示触发器的名字。例如，如果想重新编译trig\_dept\_editdata触发器，就可以使用下面的语句完成。

```
ALTER TRIGGER trig_dept_editdata COMPILE;
```

### 25.6.4 删除触发器

如果一个触发器不再使用了，可以使用DROP TRIGGER命令将其删除。删除一个特定触发器的语法规则如下：

```
DROP TRIGGER trigger_name;
```

其中，trigger\_name表示触发器的名字。例如，如果想删除trig\_dept\_editdata触发器，就可以使用下面的语句完成。

```
DROP TRIGGER trig_dept_editdata;
```

## 25.7 小结

本章主要介绍了DML触发、DDL触发器、INSTEAD OF触发器、复合触发器以及事件触发器5种类型触发器及其创建方法。通过创建触发器可以增强数据完整性和参照完整性约束，增强数据库的数据安全性。

作为继存储过程、函数之后的第三个PL/SQL命名语句块，与存储过程相比，触发器不能接收参数，不能使用EXEC语句调用，在使用触发器时，还会受到一些方面的限制。



## 第七篇 SQL应用

### 第26章 SQL语句性能优化

在实际的应用中，数据库中的数据很多，一般一个数据表中数据会有几千行、上万行甚至更多，若要从这些数据表中检索数据，就需要对系统进行优化，提高数据库系统的响应速度。提高数据库的执行效率、改善数据库的性能可以对数据库进行优化，也可以对SQL查询语句进行优化，数据库优化包括内存的分配、数据库结构的管理、数据表的设计等。除了对数据库进行优化，SQL语句的优化也是系统优化的一个很重要的方面。

SQL语句一般会消耗超过70%的数据库资源，同样的查询结果，经过优化的SQL查询语句和没有经过优化的SQL语句之间，查询数据的速度会有很大的差别，有的差别可能会达到几百倍。因此，书写SQL语句不仅要完成正确的查询功能，还应该写出高质量的SQL语句，提高系统的查询效率。本章就来介绍一些有关SQL语句查询优化的方法。

本章重点：

- ☐ 使用索引
- ☐ 在WHERE子句中适当调整连接条件的顺序
- ☐ 关键字优化
- ☐ 使用存储过程
- ☐ 规范SQL语言书写格式

#### 26.1 适当创建索引

在一般情况下，通过为数据表创建适当的索引可以加快对数据表的查询速度，提高数据库的访问性能，但同时它也会影响数据更新操作（例如插入、修改、删除）的速度。因此，创建索引并不是越多越好。在创建索引时要根据实际情况，为数据表中的适当的列创建索引。

- ☐ 如果WHERE子句中经常用到数据表中的某一列或者某几列作为查询条件，为了提高查询的效率，可以为这一列或者这一组列创建索引。
- ☐ 如果数据表中的某一列或者某几列经常需要执行排序操作，为了提高排序速度，则应该为这一列或者这一组列创建索引。



- 在多表连接时，应该为数据表的连接列创建索引。
- 对于需要频繁地进行数据更新操作（例如插入、修改、删除）的数据列，不建议在这样的数据列中创建索引。

**注意**

创建索引时，需要选择数据表中适当的列，如果数据表中的列选择不当，不但无法提高数据表的查询速度，还会对数据更新操作产生一定的影响，使得对数据的插入、修改、删除等更新操作的速度变慢。

一般情况下，可以通过为数据表适当创建索引的方法提高查询性能。但是，如果WHERE子句书写的SQL语句不合理，也会影响SQL语句的查询效率。因此在下一节将介绍在WHERE子句中的一些优化方法。

## 26.2 优化查询语句

在SQL语句中，主要使用的是WHERE子句执行查询操作，在WHERE子句中还需要经常使用一些关键字和子查询执行相关的查询操作，因此在WHERE子句中对查询的SQL语句进行优化就很重要。这一节就主要介绍如何对WHERE子句进行优化，提高数据的查询效率和性能。

### 26.2.1 避免在SELECT语句中使用“\*”

在前面的5.1节中，介绍了一种查询数据表中全部列的简便方法，就是在SELECT语句后面使用“\*”。这样虽然可以不用将要查询的数据表中的全部列一一列出，但是这却并不是一个高效的查询的方法。在使用“\*”来查询数据表中所有列的时候，数据库在解析时会通过查询数据字典将“\*”依次转换为数据表中每一列的列名，这就需要耗费额外的时间，影响查询效率。

因此，在使用SELECT语句查询时，应该在SELECT之后将需要查询的列一一列出，即使是要查询数据表中所有列的信息，也最好不要使用“\*”。从可读性方面考虑，在SELECT之后将需要查询的列一一列出，也方便用户对所要查询数据的理解。

### 26.2.2 调整WHERE子句中连接条件的顺序

在SQL语句查询语句中，使用最多的就应该是WHERE子句。对于查询大量数据的数据表来说WHERE子句后面的连接条件顺序会对数据的查询效率造成很大的影响。选择一个好的连接条件的顺序是非常重要的。下面通过一个例子来看一下WHERE子句的连接条件顺序不同对数据查询效率造成的影响。

在下面的这个例子中，要查询员工信息表中员工工资在1000到3000的员工信息。这里首先要创建一个员工信息表employee。创建员工信息表的SQL语句如下：

```
CREATE TABLE employee(  
  empno INT PRIMARY KEY,  
  eName VARCHAR (20) ,  
  salary INT,  
  eptno INT  
)
```

在employee表中一共有4列empno、eName、salary和eptno，分别用来表示员工编号、员工姓名、员

工资和员工所在的部门编号。在该表中有100000行数据，其中，empno是主键。

在employee表中，现在要查询员工信息表中员工工资在1000到3000的员工信息。可以使用两种查询方法。

第一种查询方法：

```
SELECT empno , eName, salary, deptno
FROM employee
WHERE salary>1000
AND salary<3000
```

第二种查询方法：

```
SELECT empno , eName, salary, deptno
FROM employee
WHERE salary<3000
AND salary>1000
```

这两个SQL语句的查询结果是完全相同的，但是查询的效率却是有所差别的。如果在employee表中员工的工资都在1000元以上，那么第一条SQL语句的第一个查询条件就需要占有大量的系统资源，相比较而言，第二条SQL语句的第一个查询条件就在很大程度上缩小了查询范围。也就是说，在员工的工资都在1000元以上的情况下，第二条SQL语句的效率要比第一条SQL语句的效率高。

**说明** 在使用WHERE子句进行查询时，要把限制性最高的查询条件放在最前面。

如果在WHERE子句中是含有多表连接的查询操作，那么在多表连接时，应该把表与表之间的关联字段放在前面，把相关的过滤条件放在后面。例如，要查询学生编号为s102203学生的选课成绩信息，其查询的SQL语句如下：

```
SELECT R.stuID,C.curID, C.curName,R.result
FROM T_result R,T_curriculum C
WHERE R.curID=C.curID
AND R.stuID = 's102203'
```

在这个SQL语句中，在WHERE子句里，R.curID=C.curID表示连接条件，R.stuID = 's102203'表示过滤条件。

**说明** 在多表连接时，应该将表与表的结合操作放在WHERE子句的前面，其后才是过滤条件。

### 26.2.3 避免使用OR关键字

如果想要查询与列表匹配的记录，可以有两种方式，一种是使用OR关键字，另外一种就是使用IN关键字。但是，在SQL语句中应该使用IN关键字代替OR关键字。因为OR关键字并不会使用索引，因此，使用IN关键字可以提高数据的查询速度。

例如，在表示员工信息的employee表中（在deptno列上创建了一个索引），要想查询部门编号为10和15的员工信息。如果使用OR关键字，其查询语句如下：

```
SELECT empno, eName
FROM employee
```





```
WHERE eptno =10  
OR eptno =15
```

这个SQL语句是使用OR关键字查询部门编号为10和15的员工信息，其查询所用的时间为0.1729s。如果使用IN关键字，其查询语句如下：

```
SELECT empno, eName  
FROM employee  
WHERE eptno IN(10,15)
```

使用IN关键字查询部门编号为10和15的员工信息所用的时间为0.1410s。这两个SQL语句返回的查询结果是完全一样的，但是从检索数据的速度上来看，使用IN关键字的SQL语句的检索速度要比使用OR关键字的查询语句的速度快。

#### 26.2.4 避免使用<>和!= 操作符

在前面的6.1.1小节中已经介绍了在WHERE子句中使用算术比较运算符的方法。如果想要查询教师信息表中教师工资不为3000的教师记录，读者可能很容易地就会想到使用<>或者!=完成。其查询的SQL语句如下：

```
SELECT teaID, teaName, salary  
FROM T_teacher  
WHERE salary<>3000
```

这个查询语句可以查询到用户想要的结果，但是这种查询不是使用索引，而是直接查询数据表中的数据。所以显然它的查询效率不是很高。因此，为了提高查询效率，可以把这个查询语句改写成下面这种形式。

```
SELECT teaID, teaName, salary  
FROM T_teacher  
WHERE salary<3000  
OR salary>3000
```

这个查询语句的查询效果和上面第一个查询语句的查询效果是相同的，但是这个查询语句要比上面第一个查询语句的查询速度快。所以，在实际应用中，应该尽量使用>、<、>=或者<=等操作符，而避免使用<>和!= 操作符。

#### 26.2.5 相关子查询中使用EXISTS关键字代替IN关键字

在实际应用中，如果希望将满足列表中满足指定表达式的任何一个值都查询出来，可以在WHERE语句中使用相关子查询。在WHERE子句中提供了两种相关子查询的方法，一种是使用EXISTS关键字，一种是使用IN关键字。例如，要查询选修课程编号为105这门课的学生信息，带有EXISTS关键字的相关子查询的SQL语句如下：

```
SELECT S.stuID ,S.stuName,S.age,S.sex  
FROM T_student S  
WHERE EXISTS  
(SELECT *  
FROM T_result T  
WHERE
```

```
S.stuID=T.stuID
AND T.curID = 't105')
```

这个SQL语句是使用带有EXISTS关键字的相关子查询的SQL语句查询选修课程编号为105这门课的学生信息。也可以使用带有IN关键字的相关子查询的SQL语句，其SQL语句如下：

```
SELECT S.stuID ,S.stuName,S.age,S.sex
FROM T_student S
WHERE S.stuID IN
(SELECT T.stuID
FROM T_result T
WHERE T.curID = 't105')
```

这两个SQL语句的查询结果是相同的，都是用来查询选修课程编号为105这门课的学生信息。可以看出，使用IN关键字的SQL语句书写起来比使用EXISTS关键字的SQL语句相对容易，也更容易理解，但是使用IN关键字的SQL语句的性能却没有使用EXISTS关键字的SQL语句的性能高。

因为，在执行带有IN关键字的子查询时，数据库会首先执行子查询中SQL语句，然后将查询结果放到一个临时表中。主查询会在子查询将执行完毕后（即将查询结果放到临时表之后）再执行。而带有EXISTS关键字的相关子查询的SQL语句则是首先检查主查询的第一条记录，再通过子查询检索与之匹配的数据，如果找到了，则外层查询的WHERE子句就返回TRUE，并将这条记录返回；否则返回FALSE。这个过程会反复执行，直到将外层查询的数据全部检查完毕为止。这就比带有IN关键字的子查询节省了时间，这也就是使用EXISTS关键字比使用IN关键字查询效率高的原因。

同样道理，使用NOT EXISTS关键字的SQL语句的性能要比使用NOT IN关键字的SQL语句的性能高。因此，在实际应用中，相关子查询应该使用EXISTS关键字代替IN关键字，使用NOT EXISTS关键字代替NOT IN关键字。

### 26.2.6 使用LIKE关键字

使用LIKE关键字实现模糊查询，通常它需要和通配符“%”和“\_”配合使用。通过LIKE关键字和通配符的组合，几乎可以查询任何想要查询的记录。但是，通配符如果使用不当，也会对查询的性能造成影响。

例如，要查询学生编号中含有数字100的学生信息。这需要用到LIKE关键字和通配符的组合进行模糊查询，下面给出查询学生编号中含有数字100的学生信息的两种方法。

第一种查询方法：

```
SELECT stuID,stuName,age,sex,birth
FROM T_student
WHERE stuID LIKE '%100%'
```

第二种查询方法：

```
SELECT stuID,stuName,age,sex,birth
FROM T_student
WHERE stuID LIKE 's100%'
```

这两种查询方法返回的结果都是相同的，但是其查询的性能确实不同的。第一种查询方法中，由于通配符（%）在词首出现，所以数据库不会对stuID字段使用索引，而第二种查询方法，会利用stuID

## 零基础学SQL

字段的索引进行查询，从而会大大提高查询的性能。

这里需要说明的一点是，在实际应用中，也许必须要实现第一种查询方法才能满足实际的需要，但是需要知道如果按照第一种查询方法来使用通配符，会降低查询的效率。

### 26.2.7 避免使用HAVING子句

HAVING子句是用来对分组后的结果进行过滤，限制分组后的查询结果。HAVING子句会在检索出所有记录之后才对结果集进行过滤，而且在使用HAVING子句时一般需要执行排序、统计等操作。在执行这些操作时，SQL优化器会进行一些额外的工作，这就需要消耗额外的时间。所以如果可以使用WHERE子句限制查询结果，就应该使用WHERE子句来限制查询结果，避免使用HAVING子句，也就避免让SQL优化器进行额外的开销。

### 26.2.8 使用存储过程

存储过程可以认为是经过编译的，永久保存在数据中的一组SQL语句。使用存储过程的执行效率要比使用SQL语句的执行效率高。

存储过程运行在服务器端。在存储过程运行之前，数据库已经对其进行了编译并提供了相应的优化执行方案，而SQL语句在执行时，数据库需要对其进行语法分析和检查。由于这些工作在存储过程执行之前已经完成，因此存储过程的运行性能要优于仅仅使用SQL语句的性能。

在存储过程中，SQL语句会一直保存在数据库中，随时可以被解析使用。而一般SQL语句被解析之后是保存在内存中，如果其他操作使用内存，该SQL语句就会被释放，因此它并不是永久的。

另外，使用存储过程还可以保证数据的安全性，由于存储过程中可以使用流程控制语句完成相对复杂的操作和逻辑，也使得存储过程比SQL语句有更好的灵活性。有关存储过程的内容可以参看第22章。

## 26.3 规范SQL语句书写格式

在书写SQL语句时，需要考虑SQL语句的可读性。虽然SQL的整洁性不会影响到实际的运行效率，但是书写一个可读性高的SQL语句也是十分重要的。在实际应用中，如果一个开发人员书写的SQL语句让人很难理解，那么也势必会对程序的调试增加难度。下面是两个不同SQL语句书写格式的比较。

可读性差的SQL语句：

```
SELECT T_result.stuID, T_student.stuName, T_curriculum.curID, T_curriculum.curName, T_result.result
FROM T_result ,T_curriculum ,T_student WHERE T_result.curID= T_curriculum.curID
AND T_result.stuID= T_student.stuID AND T_student.stuName = '赵亮' ORDER BY T_result.result
```

可读性好的SQL语句：

```
SELECT R.stuID,          -- 学生编号
       S.stuName,        -- 学生姓名
       C.curID,          -- 课程编号
       C.curName,        -- 课程名称
       R.result          -- 课程成绩
FROM   T_result R,       -- 成绩信息表
       T_curriculum C,   -- 课程信息表
       T_student S       -- 学生信息表
WHERE  R.curID=C.curID   -- 课程编号连接条件
```

```
AND R.stuID=S.stuID      -- 学生编号连接条件
AND S.stuName = '赵亮'   -- 学生姓名
ORDER BY R.result        -- 课程成绩
```

这两个SQL语句执行的结果是完全相同的，如果从第一个SQL语句中，很难确定这个SQL语句需要完成的是什么功能，通过第二个SQL语句就可以很清楚地知道所有查询的信息。从这两个SQL语句中，也可以总结出书写一个可读性好的SQL语句的基本规则。

- ❑ SQL语句中，每书写一个子句就需要换一行。例如SELECT子句、FROM子句、WHERE子句、ORDER BY子句等。
- ❑ 在SELECT子句中如果含有多个字段，如果需要查询的字段分别来自不同的表，则每一个字段都要使用表别名作为前缀，而且每一个字段都要另起一行。
- ❑ 在FROM子句中如果需要连接多个表，则需要为每一个表都起一个表别名，而且每一个表都要另起一行。
- ❑ 在WHERE子句中如果需要多个连接条件，则每一个连接条件都要另起一行。
- ❑ 为SQL语句添加适当的注释。

## 26.4 小结

本章主要介绍了优化SQL语句来改善数据库性能的方法。一般情况下，可以通过为数据表适当创建索引的方法提高查询性能。但是，如果WHERE子句书写的SQL语句不合理，也会影响SQL语句的查询效率。因此在本章的第二部分主要介绍的就是WHERE子句中的一些优化方法。另外，一个可读性好的SQL语句对开发人员管理和调试程序也是十分重要的，在本章的最后介绍了实际应用中SQL语句的书写规范。

SQL语句的性能优化是一个很重要也是一个很复杂的内容，这里只是在应用层次上提供了一些优化SQL改善性能的方法，数据库性能优化还涉及到网络层的流量控制、底层数据库的资源配置等多个方面。如果希望更深入地了解数据库的性能优化，可以查看相关书籍。

## 第27章 动态SQL

在前面介绍的PL/SQL语句块中，PL/SQL使用早期绑定执行SQL语句。即执行的SQL在编译期间就已经确定下来，它不会随着程序执行而发生变化，这样的SQL语句称为静态SQL语句。但是在实际应用中，有些时候需要在程序运行时建立并处理相应的SQL语句，即在执行PL/SQL语句块中动态地处理SQL语句。

当执行DDL语句（例如，CREATE、DROP等）、DCL语句（例如，CREATE、ROVOKE等）或者是会话控制语句（例如，ALTER SESSION等）时，需要使用动态SQL。为了在程序中可以更灵活地执行SELECT语句或者DML语句，例如，有些时候有可能直到程序运行时才会知道DML语句中相关的模式名、表名和列名，这时也需要使用动态SQL语句。

在PL/SQL中提供了两种方式来实现在动态SQL语句。一种方式是使用本地动态SQL，另一种方式是使用DBMS\_SQL包。本章就来介绍实现动态SQL语句的方法。

本章重点：

- ☐ 使用EXECUTE IMMEDIATE语句处理单行查询
- ☐ 使用游标变量处理多行查询
- ☐ 批量绑定
- ☐ 使用DBMS\_SQL包实现动态SQL

### 27.1 使用EXECUTE IMMEDIATE语句处理单行查询

EXECUTE IMMEDIATE语句可以用来处理DDL语句（例如，CREATE、DROP等）、DCL语句（例如，CREATE、ROVOKE等）、DML语句（例如，INSERT、UPDATE等）以及SELECT INTO单行查询语句。其语法规则如下：

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... | record}]
[USING [IN | OUT | IN OUT] bind_argument[, [IN | OUT | IN OUT] bind_argument]...]
[{RETURNING | RETURN} INTO bind_argument[, bind_argument]...];
```

其中，dynamic\_string表示一条有效的SQL语句或者是一个PL/SQL语句块的字符串变量。字符串中可以包括用于参数绑定的占位符；define\_variable表示存放被选出的字段值的变量；record表示存放被选出的行记录，其记录类型为用户定义类型或者是%ROWTYPE类型；bind\_argument表示参数，其参数模式可以是输入模式、输出模式或者是输入输出模式。一个参数模式为输入模式的bind\_argument参数是一个表达式，而一个参数模式为输出模式的bind\_argument参数就是一个能保存动态SQL返回值的变量。

INTO子句用来指定存放检索值的变量或者记录。对于检索出来的每一个值，在INTO子句中都需要



有一个与检索出来的值相对应的、类型兼容的变量或者是字段。USING子句用来为数据绑定的占位符提供输入数据。每个占位符需要对应于USING子句中的一个绑定参数。绑定参数可以是数字、字符、字符串。

RETURNING INTO子句用来接收DML语句返回的变量或者记录，在RETURNING INTO之后也需要使用占位符。同USING子句一样，每个占位符需要对应于RETURNING INTO子句中的一个绑定参数。绑定参数可以是数字、字符、字符串。对于含有RETURNING子句的DML语句来说，其默认的参数模式为OUT。如果在EXECUTE IMMEDIATE语句中同时使用了USING子句和RETURNING INTO子句，那么USING子句中的参数模式只能使用IN模式。

下面通过一个例子来看一下如何使用EXECUTE IMMEDIATE语句处理DDL语句、DML语句、PL/SQL语句块和SELECT INTO单行查询语句。

```
DECLARE
    sql_executeTest    VARCHAR2(100);
    plsql_block        VARCHAR2(500);
    executeTest_id     NUMBER(4) := 10;
    executeTest_alary   NUMBER(7, 2) := 3000;
    executeTest_name    VARCHAR2(14) := 'MARK';
    v_name             executeTest_table.name %ROWTYPE;
    executeTest_record  executeTest_table %ROWTYPE;
BEGIN
    /*处理DDL语句，创建executeTest_table表*/
    EXECUTE IMMEDIATE 'CREATE TABLE executeTest_table(id INT,ename VARCHAR2, salary NUMBER)';
    sql_executeTest:= 'INSERT INTO executeTest_table VALUES (:id, :ename, : salary)';
    /*处理DML语句，向executeTest_table表中插入数据*/
    EXECUTE IMMEDIATE sql_executeTest
        USING executeTest_id, executeTest_name, executeTest_salary;
    plsql_block :=
        'DECLARE
            CURSOR cursor_executeTest ISSELECT * FROM executeTest_table
        BEGIN
            FOR executeTest_record IN cursor_executeTest LOOP
                DBMS_OUTPUT.PUT_LINE ('id: ' || executeTest_record.id
                    || 'name: ' || executeTest_record.name
                    || 'salary: ' || executeTest_record.salary);
            END';
    /*处理PL/SQL语句块*/
    EXECUTE IMMEDIATE plsql_block;
    sql_executeTest:= 'SELECT * FROM executeTest_table WHERE executeTestno = :id';
    /*使用USING子句绑定参数*/
    EXECUTE IMMEDIATE sql_executeTest
        INTO executeTest_record
        USING executeTest_id;
    sql_executeTest:='DELETE executeTest WHERE id = :10 RETURNING ename INTO : name ';
    /*使用RETURNING子句接收DML语句返回的值*/
    EXECUTE IMMEDIATE sql_executeTest
        RETURNING INTO v_name;
    /*处理DDL语句，删除executeTest_table表*/
```

```
EXECUTE IMMEDIATE 'DROP TABLE executeTest_table';  
END;
```

在这段PL/SQL语句块中演示了使用EXECUTE IMMEDIATE语句处理DDL语句、DML语句、PL/SQL语句块和SELECT INTO单行查询语句的方法。EXECUTE IMMEDIATE执行的字符串既可以是字符串（例如，CREATE TABLE executeTest\_table语句和DROP TABLE executeTest\_table），也可以是一条SQL语句（例如，INSERT INTO executeTest\_table），还可以是一个PL/SQL语句块的字符串变量（例如，plsql\_block语句块）。

可以为绑定变量重新指定新值执行动态SQL语句。但是由于EXECUTE IMMEDIATE语句在每次执行之前都会对动态字符串进行预处理操作，所以这样做会消耗大量的资源。

**注意** 动态SQL支持所有的SQL类型，但是动态SQL不支持PL/SQL特有的类型。例如，在动态SQL中不能使用布尔型和索引表。

## 27.2 使用游标变量处理多行查询

在前面一节中讲到的EXECUTE IMMEDIATE语句只能动态地处理单行查询，而不能处理多行查询。为了可以动态地处理多行查询，就需要使用OPEN-FOR、FETCH和CLOSE游标操作语句来完成。其中，OPEN语句用于打开游标变量，FETCH语句用于从游标结果集中提取数据，CLOSE语句用于关闭游标变量。本节就来介绍使用游标变量处理多行查询的方法。

### 27.2.1 定义游标变量

为了使用游标变量动态地处理多行查询，首先就需要定义游标变量。定义游标时需要指定游标的类型，并要为游标定义一个名字。定义游标的语法规则如下：

```
TYPE cursortype IS REF CURSOR;  
cursor_variable cursortype;
```

其中，cursortype用来执行游标的类型为REF CURSOR，cursor\_variable表示游标的名字。例如，要定义一个executeTestcurtyp游标，就可以使用下面的语句。

```
TYPE executeTestcurtyp IS REF CURSOR;  
executeTest_cv executeTestcurtyp;
```

### 27.2.2 打开游标变量

在定义完一个游标之后，就可以使用OPEN-FOR语句打开这个游标了。OPEN-FOR语句可以把游标变量和一个多行查询语句关联在一起。只有在打开了游标之后，才可以执行相关数据提取操作。打开游标的语法规则如下：

```
OPEN cursor_variable FOR dynamic_string  
[USING bind_argument[, bind_argument]...];
```

其中，cursor\_variable表示前面已经定义的游标的名字，dynamic\_string是一个字符串表达式，用于指定多行查询语句；bind\_argument用来表示为占位符传递数据的变量。USING子句是可选的。程序运

行时，动态SELECT语句中相对应的占位符可以用USING子句中的绑定变量替换掉。

下面来看一个打开游标变量的例子。在下面的这个例子中，首先定义一个游标变量，然后将它和一个动态的SELECT语句关联在一起。

```
DECLARE
    TYPE executeTestcurtyp IS REF CURSOR;    --定义游标
    executeTest_cv executeTestcurtyp;
    tea_teaName VARCHAR2(10);
    tea_salary NUMBER := 3500;
BEGIN
    OPEN executeTest_cv FOR                  --打开游标
        'SELECT teaName , salary
        FROM t_teacher
        WHERE salary > :s'
        USING tea_salary;
END;
```

在这个PL/SQL语句块中，使用OPEN-FOR语句将游标变量executeTestcurtyp和一个多行查询语句关联在一起。通过OPEN-FOR语句打开游标变量，可以执行多行查询语句的查询操作，并确定查询的结果集，将游标放在结果集的第一行，并把%ROWCOUNT值初始化为零。

### 27.2.3 从游标变量取得数据

在完成了声明游标和打开游标的操作之后，就可以使用游标取得结果集中的数据了。使用游标取得结果集中的数据使用FETCH语句来完成。其语法规则如下：

```
FETCH cursor_variable
INTO {define_variable[, define_variable]... | record};
```

其中，cursor\_variable表示前面已经定义的游标的名字；define\_variable用来表示接收数据的变量；record用来表示接收数据的记录变量。

FETCH语句可以从查询的结果集中返回一个单行数据，并将得到的数据值赋值给INTO子句后对应的变量，赋值操作完成之后，属性%ROWCOUNT的值会自动加1，这样游标就会移动到下一行，继续查询下一行的数据值。依次类推，直达将所有数据全部都查询出来为止。

例如，对于在前面定义的游标变量executeTest\_cv，如果希望将游标变量executeTest\_cv中取得的数据放到变量tea\_teaName和tea\_salary中，可以使用下面的语句来完成。

```
LOOP
    FETCH executeTest_cv
        INTO tea_teaName, tea_salary;
    EXIT WHEN executeTest_cv%NOTFOUND;
    --执行操作代码
END LOOP;
```

当然，也可以将游标变量executeTest\_cv中取得的数据放到一个记录中。如果希望将游标变量executeTest\_cv中取得的数据放到一个记录中，可以使用下面的语句来完成。

```
DECLARE
    tea_record t_teacher%ROWTYPE;
```

```
BEGIN
  LOOP
    FETCH executeTest_cv INTO tea_record;
    EXIT WHEN executeTest_cv%NOTFOUND;
    -- 执行操作代码
  END LOOP;
END;
```

#### 27.2.4 关闭游标变量

当查询的结果集检索完成之后，就可以关闭游标了。关闭游标是使用CLOSE语句来完成的。关闭游标后，与游标相关的资源将会被释放。关闭游标的语法规则如下：

```
CLOSE cursor_variable;
```

其中，cursor\_variable表示关闭游标的名字。这里的cursor\_variable是前面已经打开的游标名。游标被关闭后还可以重新将其打开。

例如，对于在前面定义的游标变量executeTest\_cv，如果所有的数据都处理完毕，就可以使用CLOSE语句将其关闭了。

```
CLOSE executeTest_cv;
```

如果一个游标关闭之后，还要使用它来检索数据，那么Oracle数据库管理系统会给出下面的一个错误提示。

```
ORA-1001:Invalid Cursor
```

同样，如果要对一个已经关闭了的游标进行操作也是非法的，此时会Oracle数据库管理系统抛出INVALID\_CURSOR异常。

#### 27.2.5 动态处理多行查询

在前面的几节中介绍了游标定义、打开游标、从游标中提取数据以及关闭游标变量的方法，本节通过一个例子来看一下如何使用游标动态地处理多行数据。

```
DECLARE
  TYPE executeTestcurtyp IS REF CURSOR;           -- 定义游标
  executeTest_cv executeTestcurtyp;
  sql_stmt VARCHAR2(200);
  tea_ teaName VARCHAR2(10);
  tea_ salary  NUMBER := 3500;
  tea_record t_teacher%ROWTYPE;
BEGIN
  sql_stmt := 'SELECT teaName , salary
               FROM t_teacher
               WHERE salary > :s'
  OPEN executeTest_cv FOR sql_stmt                -- 打开游标
  USING tea_ salary;
  LOOP
    FETCH executeTest_cv                          -- 提取数据
    INTO tea_record;
```



```
EXIT WHEN executeTest_cv%NOTFOUND;
DBMS_OUTPUT.PUT_LINE ('教师姓名: ' || tea_record.teaName
                      || '工资: ' || tea_record.salary);
END LOOP;
CLOSE executeTest_cv;          -- 关闭游标
END;
```

在这段PL/SQL语句块中，使用游标变量动态处理多行查询。根据教师工资水平，将教师工资在3500以上的教师姓名及其对应的工资查询出来。通过FETCH语句从游标变量中提取数据，将其放到一个记录中，并将查询到的教师姓名及其对应的工资显示输出。在所有的数据都处理完毕之后，使用CLOSE语句将游标变量executeTest\_cv关闭。最后查询的结果如下所示。

```
教师姓名: 张昌 工资: 3800
教师姓名: 毛翠 工资: 4000
教师姓名: 李中 工资: 4200
```

## 27.3 批量绑定

批量绑定是指把SQL语句中的一个变量与一个集合绑定在一起。集合元素必须是SQL数据类型（例如，CHAR、DATE、NUMBER等）。在动态SQL中使用批量绑定可以加快批量数据的处理速度，减少PL/SQL和SQL引擎之间的切换，提高PL/SQL程序应用性能。支持动态批量绑定的语句主要有FORALL、EXECUTE IMMEDIATE和FETCH三种。本节就来介绍使用这三种语句实现批量绑定的方法。

### 27.3.1 使用FORALL语句批量绑定

FORALL语句批量绑定允许在动态SQL语句中批量绑定输入参数。可以在动态DML上使用FORALL和EXECUTE IMMEDIATE语句实现动态绑定，其语法规则如下：

```
FORALL index IN lower bound..upper bound
EXECUTE IMMEDIATE dynamic_string
USING bind_argument | bind_argument(index)
[, bind_argument | bind_argument(index)] ;
```

其中，dynamic\_string是一个字符串表达式，该字符串必须是一个INSERT、UPDATE或者DELETE语句；USING子句中的绑定变量用来替换动态SELECT语句中相对应的占位符；bind\_argument用来表示为占位符传递数据的变量。下面通过一个例子来看一下如何在动态DML上使用FORALL和EXECUTE IMMEDIATE语句实现动态绑定。

```
DECLARE
TYPE teaID_table_type IS TABLE OF t_teacher.teaID%TYPE;
teaID_tab teaID_table_type;
sql_stmt VARCHAR2(200);
BEGIN
teaID_tab:= teaID_table_type ('t156354', 't186585', 't105320');
sql_stmt := 'UPDATE t_teacher SET salary= salary * 1.2 WHERE teaID = :tID ';
FORALL i IN 1 .. teaID_tab.COUNT
EXECUTE IMMEDIATE sql_stmt
USING teaID_tab (i);
```



END;

这段PL/SQL语句是用来增加指定教师的工资。这里在DECLARE部分定义了一个嵌套表teaID\_tab，在BEGIN部分，为这个嵌套表赋了3个值。在FORALL语句中，使用USING子句绑定的嵌套表teaID\_tab中的值替换掉动态UPDATE语句中相对应的占位符tID。

### 27.3.2 使用EXECUTE IMMEDIATE语句批量绑定

使用EXECUTE IMMEDIATE语句可以把变量或者OUT模式的参数批量绑定到一个动态的SQL语句，其语法规则如下：

```
EXECUTE IMMEDIATE dynamic_string
  [[BULK COLLECT] INTO define_variable[, define_variable ...]]
  [[USING bind_argument[, bind_argument ...]]]
  [{RETURNING | RETURN}
  BULK COLLECT INTO bind_argument[, bind_argument ...]];
```

其中，dynamic\_string是一个字符串表达式，该字符串必须是一个INSERT、UPDATE或者DELETE语句；USING子句是可选的，USING子句中的绑定变量用来替换动态SELECT语句中相对应的占位符；BULK COLLECT INTO子句用来绑定变量；RETURNING BULK COLLECT INTO子句用来批量绑定输出变量。

**注意** 如果在BULK COLLECT INTO子句中的变量个数超过查询列的个数，则Oracle会产生一个错误。

下面通过一个例子来看一下如何使用EXECUTE IMMEDIATE语句实现动态绑定。这个例子是在DML返回子句中使用批量绑定，提高指定院系的教师工资。

```
DECLARE
  TYPE deptName_table_type IS TABLE OF t_teacher. deptID %TYPE;
  deptName_tab deptName_table_type;
  TYPE teaSalary_table_type IS TABLE OF t_teacher.salary %TYPE;
  teaSalary_tab teaSalary_table_type;
  sql_stmt VARCHAR2(200);
  deptID VARCHAR (15) := 't_10';
  tName t_teacher. teaName %TYPE;
  tSalary t_teacher. salary%TYPE;
BEGIN
  sql_stmt := 'UPDATE t_teacher SET salary= salary * 1.2 WHERE deptID = : dID ';
  RETURNING teaName, salary INTO : tName: tSalary;
  EXECUTE IMMEDIATE sql_stmt
  USING deptID
  RETURNING BULK COLLECT INTO deptName_tab, teaSalary_tab;
  FOR i IN 1 .. deptID_tab.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE ('教师姓名: ' || deptName_tab(i)
                          || '新工资: ' || teaSalary_tab(i);
  END LOOP;
END;
```

在这段PL/SQL语句块中，在DECLARE部分定义了两个嵌套表deptID\_tab和teaSalary\_tab。在BEGIN部分，使用USING子句绑定的变量deptID的值替换掉动态UPDATE语句中相对应的占位符dID。

RETURNING BULK COLLECT INTO子句批量绑定了两个嵌套表变量。最后使用FOR循环语句将两个嵌套表deptID \_tab和teaSalary \_tab中的值一一显示输出。其显示结果如下所示。

```
教师姓名: 张昌 新工资: 4560
教师姓名: 赵伟 新工资: 3600
教师姓名: 毛翠 新工资: 4800
教师姓名: 于波 新工资: 3360
```

### 27.3.3 使用FETCH 语句批量绑定

使用FETCH 语句批量绑定允许从动态游标中取得数据，如果希望在FETCH语句中取得多条记录，可以使用BULK COLLECT子句。BULK COLLECT子句可以批量绑定数据。使用FETCH语句批量绑定的语法规则如下：

```
FETCH cursor_variable
BULK COLLECT INTO define_variable[, define_variable ...];
```

其中，cursor\_variable表示游标变量名；define\_variable表示接收查询结果的集合变量。下面通过一个例子来看一下FETCH 语句批量绑定的用法。

```
DECLARE
    TYPE executeTestcurtyp IS REF CURSOR;           -- 定义游标
    executeTest_cv executeTestcurtyp;
    TYPE profession_table_type IS TABLE OF t_teacher. profession %TYPE;
    profession_tab profession_table_type;
    TYPE teaName_table_type IS TABLE OF t_teacher. teaName %TYPE;
    teaName_tab teaName_table_type;
    deptID VARCHAR (15) := 't_10';
    sql_stmt VARCHAR2(200);
BEGIN
    sql_stmt := 'SELECT teaName , profession
                FROM t_teacher
                WHERE deptID :=:dID;
    OPEN executeTest_cv FOR sql_stmt                -- 打开游标
    USING deptID;
    FETCH executeTest_cv                            -- 提取数据
    BULK COLLECT INTO teaName_tab t,profession_tab;
    CLOSE executeTest_cv;                           -- 关闭游标
    FOR i IN 1 .. teaName_tab.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE ('教师姓名: ' || teaName_tab(i)
                               || '职称: ' || profession_tab(i));
    END LOOP;
END;
```

在这段PL/SQL语句块中，在DECLARE部分定义了一个游标变量executeTestcurtyp、两个嵌套表teaName\_tab和profession\_tab。在BEGIN部分，使用OPEN...FOR语句打开游标，并使用USING子句绑定的变量deptID的值替换掉动态UPDATE语句中相对应的占位符dID。使用FETCH语句从游标变量中提取数据，通过BULK COLLECT INTO语句绑定两个嵌套表teaName\_tab和profession\_tab，然后使用CLOSE语句关闭游标。最后使用FOR循环语句将两个嵌套表teaName\_tab和profession\_tab中的值一一显

示输出。其显示结果如下所示。

```
教师姓名: 张昌  职称: 教授
教师姓名: 赵伟  职称: 副教授
教师姓名: 毛翠  职称: 教授
教师姓名: 于波  职称: 讲师
```

## 27.4 使用DBMS\_SQL包实现动态SQL

在PL/SQL程序设计过程中，很多时候需要使用动态SQL解决问题。除了前面介绍的使用本地动态SQL之外，在Oracle系统中还提供了一个专门用于在PL/SQL中动态执行SQL的系统包——DBMS\_SQL包。相对于本地动态SQL会在每次执行时都需要进行解析，DBMS\_SQL可以分析一次执行多次进行。本节就来介绍DBMS\_SQL包中常用存储过程和函数以及如何使用DBMS\_SQL包实现动态SQL。

### 27.4.1 DBMS\_SQL包中常用存储过程和函数

在DBMS\_SQL系统包中，包括了执行打开游标、分析语句、变量绑定、关闭游标等操作的很多的存储过程和函数，通过这些存储过程和函数，可以有效地实现对SQL语句进行控制和处理。下面介绍一些在DBMS\_SQL系统包中经常用到的存储过程和函数。

- ❑ **FUNCTION OPEN\_CURSOR**: 打开指定的动态游标。函数返回游标ID的值，其返回值类型为INTEGER类型。
- ❑ **PROCEDURE PARSE**(c IN INTEGER, statement IN VARCHAR2, language\_flag IN INTEGER): 解析动态游标中的SQL语句。其中，参数c用来指定游标ID；参数statement表示被解析的SQL语句；参数language-flag表示解析SQL语句使用的Oracle版本（包括V6、V7和NATIVE 3个版本）。
- ❑ **PROCEDURE BIND\_VARIABLE**(c IN INTEGER, name IN VARCHAR2, value): 绑定游标结果到指定变量。其中，参数c用来指定游标ID；参数name表示变量名称；参数value表示绑定到变量的值。
- ❑ **PROCEDURE BIND\_VARIABLE\_CHAR**(c IN INTEGER, name IN VARCHAR2, value, [column\_size IN INTEGER]): 绑定游标结果到指定变量。其中，参数c用来指定游标ID；参数name表示变量名称；参数value表示绑定到变量的值；参数column\_size表示指定列的长度，该参数只有在CHAR、VARCHAR2等类型中使用。它是可选的。
- ❑ **FUNCTION EXECUTE**(c IN INTEGER): 使用游标执行语句。函数返回值为一个INTEGER类型的值。对于DML语句（INSERT、UPDATE、DELETE），如果返回1，则表示执行成功；如果返回0，则表示执行失败。其中，参数c用来指定游标ID。
- ❑ **FUNCTION FETCH\_ROWS**(c IN INTEGER): 获取游标结果中的数据。函数返回值为一个INTEGER类型的值。如果返回值为0，则表示已经取值取到了游标最后。
- ❑ **PROCEDURE COLUMN\_VALUE**(c IN INTEGER, position IN INTEGER, value): 取得一个游标在指定位置上对应的变量值。其中，参数c用来指定游标ID；position表示指定字段在游标中的位置（从1开始）；value表示绑定到变量的值。（该过程需要在函数FETCH\_ROWS之后调用）
- ❑ **PROCEDURE DEFINE\_COLUMN**(c IN INTEGER, position IN INTEGER, column IN datatype, [column\_size IN INTEGER]): 定义动态游标获取的被选择列的对应值。其中，参数c用来指定游

标ID；position表示指定字段在游标中的位置（从1开始）；参数column指定被定义的列；参数column\_size表示指定列的长度，该参数只有在CHAR、VARCHAR2等类型中使用。它是可选的。

□ PROCEDURE CLOSE\_CURSOR(c IN OUT INTEGER)：关闭指定的动态游标。其中，参数c用来指定游标ID。

在介绍存储过程PARSE的时候，提到了一个Oracle的版本。Oracle的版本包括V6、V7和NATIVE这3个版本。这3个版本在DBMS\_SQL系统包中被定义为INTEGER类型的常量。

```
V6 CONSTANT INTEGER := 0;
V7 CONSTANT INTEGER := 1;
NATIVE CONSTANT INTEGER := 2;
```

**说明** 在不清楚所连接的Oracle版本时，可以使用NATIVE。

### 27.4.2 使用DBMS\_SQL包实现动态SQL的查询操作

如果希望使用DBMS\_SQL包实现动态SQL的查询操作，一般需要经历打开游标、解析SQL、绑定变量、获取被选择列的对应值、执行SQL、获取查询结果值以及关闭游标等几个步骤。下面通过一个例子来看一下如何使用DBMS\_SQL包实现动态SQL的查询操作。

```
CREATE OR REPLACE PROCEDURE get_teacherInfo
(p_deptID IN t_teacher.deptID% TYPE DEFAULT NULL )
AS
    v_cursorID INTEGER;                -- 游标ID
    v_select varchar2(200);            -- 查询SQL语句
    v_teaID VARCHAR (15);              -- 教师编号
    v_teaName VARCHAR (10);            -- 教师姓名
    v_dept VARCHAR (20);               -- 教师所在院系
    v_selectResult INTEGER;            -- 游标执行语句返回结果
BEGIN
    v_cursorID:= DBMS_SQL.OPEN_CURSOR; -- 打开游标
    v_select:=SELECT teaID, teaName , dept FROM t_teacher WHERE deptID'= :deptID';
    DBMS_SQL.PARSE (v_cursorID,v_updateResult, DBMS_SQL.NATIVE); -- 解析SQL
    DBMS_SQL.BIND_VARIABLE_CHAR (v_cursorID, :deptID,p_deptID); -- 绑定变量
    DBMS_SQL.DEFINE_COLUMN (v_cursorID,1,v_teaID,15); -- 获取的被选择列的对应值
    DBMS_SQL.DEFINE_COLUMN (v_cursorID,2,v_teaName,10);
    DBMS_SQL.DEFINE_COLUMN (v_cursorID,3,v_dept,20);
    v_row selectResult Updated:= DBMS_SQL.EXECUTE (v_cursorID); -- 执行SQL
    LOOP
        IF DBMS_SQL.FETCH_ROWS (v_cursorID)=0 THEN
            EXIT;
        END IF;
        DBMS_SQL.COLUMN_VALUE (v_cursorID,1,v_teaID); -- 取得指定位置对应的值
        DBMS_SQL.COLUMN_VALUE (v_cursorID,2,v_teaName);
        DBMS_SQL.COLUMN_VALUE (v_cursorID,3,v_dept);
    END LOOP;
    DBMS_SQL.CLOSE_CURSOR (v_cursorID); -- 关闭游标
    COMMIT; --提交事务
EXCEPTION
```



## 零基础学SQL

```
WHEN OTHERS THEN
    DBMS_SQL. CLOSE_CURSOR (v_ cursorID);           -- 关闭游标
    RAISE;
END;
```

这里创建了一个名为get\_teacherInfo的存储过程。在这个存储过程中接收一个表示院系编号的p\_deptID参数，根据院系编号查询教师信息。在BEGIN部分，首先通过OPEN\_CURSOR函数打开游标，并将该函数的返回值赋值给v\_ cursorID。然后使用PARSE过程解析查询教师信息的SQL语句，并通过BIND\_VARIABLE\_CHAR过程绑定变量deptID。在使用DEFINE\_COLUMN过程获取的被选择列的对应值之后，通过EXECUTE函数执行查询的SQL语句。

在LOOP循环中，FETCH\_ROWS函数用来判断是否已经取值到了游标的末尾。如果函数返回的结果为0，则表示数据已经全部取出了，此时使用EXIT退出程序。如果函数返回的结果不为0，则会使用COLUMN\_VALUE过程将指定位置对应结果值一一取出。在结束LOOP循环之后，使用CLOSE\_CURSOR过程关闭游标。

在EXCEPTION部分，通过WHEN OTHERS THEN子句对所有可能发生的异常进行处理，并关闭游标，以防止资源泄露。

### 27.4.3 使用DBMS\_SQL包实现动态SQL的DML操作

如果希望使用DBMS\_SQL包实现动态SQL的INSERT和UPDATE操作，一般需要经历打开游标、解析SQL、绑定变量、执行SQL以及关闭游标等几个步骤。下面通过一个例子来看一下如何使用DBMS\_SQL包实现动态SQL的DML操作。

```
CREATE OR REPLACE FUNCTION updat_result (
    (p_ result IN t_result. result %TYPE
    p_stuID IN t_ t_result.stuID%TYPE,
    p_curID IN t_result.curID %TYPE
    )
    RETURN INTEGER
    AS
        v_cursorID INTEGER;           -- 游标ID
        v_rowUpdated INTEGER;         -- 游标执行语句返回结果
        v_updateResult VARCHAR2 (100); -- 修改成绩表中数据的SQL语句
        v_ result INT;               -- 学生成绩
    BEGIN
        v_ cursorID:= DBMS_SQL. OPEN_CURSOR; -- 打开游标
        v_ updateResult:='UPDATE t_result SET result =: result -- 修改学生课程成绩的SQL语句
                        WHERE stuID = :stuID AND curID = :curID';
        DBMS_SQL. PARSE (v_ cursorID,v_ updateResult, DBMS_SQL. NATIVE); -- 解析SQL
        DBMS_SQL. BIND_VARIABLE (v_ cursorID, :result,p_ result ); -- 绑定变量
        DBMS_SQL. BIND_VARIABLE_CHAR (v_ cursorID, :stuID,p_ stuID);
        DBMS_SQL. BIND_VARIABLE_CHAR (v_ cursorID, : curID,p_ curID);
        v_rowUpdated:= DBMS_SQL. EXECUTE (v_ cursorID); -- 执行SQL
        DBMS_SQL. CLOSE_CURSOR (v_ cursorID); -- 关闭游标
    RETURN p_rowUpdated;
    EXCEPTION
    WHEN OTHERS THEN
```



```
DBMS_SQL. CLOSE_CURSOR (v_ cursorID);          -- 关闭游标
RAISE;
END;
```

这里创建了一个名为updat\_result的函数。在这个函数中需要接收3个参数，分别用来表示院系编号、学生课程成绩、学生编号和课程编号。在BEGIN部分，首先通过OPEN\_CURSOR函数打开游标，并将该函数的返回值赋值给v\_cursorID。然后使用PARSE过程解析UPDATE语句，并通过BIND\_VARIABLE过程绑定变量result，通过BIND\_VARIABLE\_CHAR过程绑定变量stuID和curID。之后通过EXECUTE函数执行UPDATE语句。最后关闭游标。

在EXCEPTION部分，通过WHEN OTHERS THEN子句对所有可能发生的异常进行处理，并关闭游标，以防止资源泄露。

**说明** 如果希望使用DBMS\_SQL包实现动态SQL的DELETE操作，只需要经历打开游标、解析SQL、执行SQL以及关闭游标等几个步骤就可以了。

#### 27.4.4 使用DBMS\_SQL包实现动态SQL的DDL操作

使用DBMS\_SQL包实现动态SQL的DDL操作与前面介绍的实现动态SQL的查询操作和实现动态的DML操作有所不同，在使用DBMS\_SQL包实现动态SQL的DDL操作时，DDL语句在解析后会立即得到执行，并不需要调用EXECUTE过程。下面通过一个例子来看一下如何使用DBMS\_SQL包实现动态SQL的DDL操作。

**注意** 在DDL语句中，不能使用BIND\_VARIABLE过程绑定变量。

```
CREATE OR REPLACE PROCEDURE creatTable
(tablename VARCHAR2,
columns VARCHAR2)
AS
    v_ cursorID INTEGER;          -- 游标ID
    v_createTable varchar2(100);  -- 创建数据表SQL语句
BEGIN
    v_ cursorID:= DBMS_SQL. OPEN_CURSOR;          -- 打开游标
    v_createTable = 'CREATE TABLE(tablename, columns)';
    DBMS_SQL. PARSE (v_ cursorID,v_ createTable, DBMS_SQL. NATIVE);  -- 解析SQL
    DBMS_SQL. CLOSE_CURSOR (v_ cursorID);        -- 关闭游标
EXCEPTION
    WHEN OTHERS THEN
        DBMS_SQL. CLOSE_CURSOR (v_ cursorID);    -- 关闭游标
    RAISE;
END;
```

这里创建了一个名为creatTable的存储过程。在这个函数需要接收两个参数，分别用来表示创建表的两个列名。在BEGIN部分，首先通过OPEN\_CURSOR函数打开游标，并将该函数的返回值赋值给v\_cursorID。然后使用PARSE过程解析CREATE TABLE语句，解析之后，使用CLOSE\_CURSOR过程，最后关闭游标即可。在解析之后，该CREATE TABLE语句就会执行。

## 零基础学SQL

在EXCEPTION部分，通过WHEN OTHERS THEN子句对所有可能发生的异常进行处理，并关闭游标，以防止资源泄露。

### 27.5 小结

本章主要介绍了动态SQL的实现方法。使用EXECUTE IMMEDIATE语句可以用来处理大多数的动态SQL语句，如果需要处理多行查询的动态SQL语句，就需要使用REF CURSOR类型的游标变量来处理。使用游标变量需要用到OPEN-FOR语句、FETCH语句和CLOSE语句。其中，OPEN语句用于打开游标变量，FETCH语句用于从游标结果集中提取数据，CLOSE语句用于关闭游标变量。

另外，还介绍了实现批量绑定的3种方法：FOR ALL、EXECUTE IMMEDIATE和FETCH。在本章的最后介绍了使用DBMS\_SQL包实现动态SQL的查询、DML和DDL操作方法。使用DBMS\_SQL包实现动态SQL可以分析一次多次执行，而使用本地动态SQL则需要在每次执行时都要进行分析。但是与DBMS\_SQL相比，EXECUTE IMMEDIATE实现动态SQL也相对容易。

## 第28章 数据库的存取访问

大部分的实际应用程序中，都需要在一定的开发环境下使用程序设计语言通过SQL语句对数据库中的数据进行存取操作。这就需要程序设计语言可以对数据库进行连接和访问。本章将从数据库应用系统结构入手，介绍数据库应用系统结构的4种基本结构，然后介绍几种常用的数据库连接访问技术，最后通过一种高级程序设计语言Java与一个数据库MySQL 5.0的连接和开发的例子介绍如何使用程序设计语言实现对数据库的连接和访问。

本章重点：

- ☐ 数据库应用系统结构
- ☐ 数据库连接访问技术
- ☐ Java与MySQL 5.0数据库连接与访问
- ☐ Java与MySQL 5.0数据库开发

### 28.1 数据库应用系统结构

一个数据库应用系统中，其系统结构一般包括界面显示层、业务逻辑层和数据处理层3个部分。其中，界面显示层主要用于为用户操作提供可视化的界面，方便用户对数据进行请求和处理；业务逻辑层主要用于处理与用户操作相关的各种业务处理，通常由程序设计语言来完成；数据处理层主要用于处理和维持各种数据信息，主要由数据库管理系统来完成。目前的数据库应用系统中主要包括集中式数据库系统、客户端/服务器端数据库系统、并行式数据库系统和分布式数据库系统这4种基本的数据库应用系统结构。

#### 28.1.1 集中式数据库系统

集中式数据库系统（Centralized DBS）是指在单机上运行的数据库管理系统。集中式数据库系统可以用于单个用户，也可以用于多个用户。

在单个用户的集中式数据库系统中，界面显示层、业务逻辑层和数据处理层3个部分都在一台个人计算机上完成，这样的数据库管理系统也叫做桌面数据库管理系统。目前常用的桌面数据库管理系统包括Access数据库和Visual FoxPro数据库。单个用户的集中式数据库系统不支持并发控制，在数据完全性、完整性和数据一致性等方面还有很多不足。

在多个用户的集中式数据库系统中，有多个计算机终端通过一个数据链与主机服务器相连，可以为多个不同用户提供服务。同时，在多用户的DBS中，利用操作系统提供的多任务机制处理，可以用于单机分时系统的开发环境，允许并发执行多个查询操作。

## 28.1.2 客户端/服务器端数据库系统

客户端/服务器端数据库系统（Client/Server DBS或者C/S DBS）可以用于计算机网络环境，可以通过网络进行数据访问。其中，每一个用户的个人计算机作为客户端，主机作为服务器端，主机与客户端之间通过网络完成数据通信。C/S结构如图28.1所示。

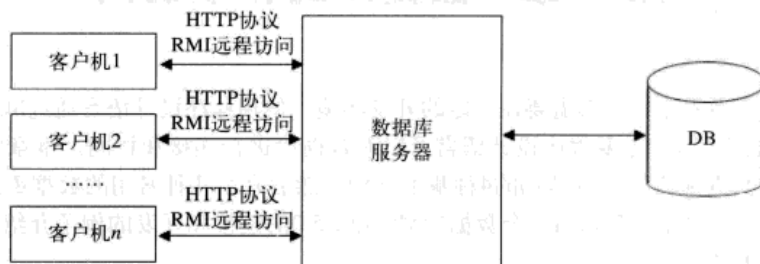


图28.1 两层C/S结构

在客户端/服务器端数据库系统中，客户机主要用来完成应用程序的处理。例如，界面的显示、数据的输入输出、显示格式的处理等。主机主要用来完成对数据库管理系统（DBMS）和操作系统中数据的管理。例如，数据查询、数据表的存取结构、系统恢复、数据的并发控制等。

目前大多数的软件应用系统采用的都是C/S结构，通过使用客户端/服务器端数据库系统可以将任务合理地分配到客户端，这种功能的分布降低了计算机系统的开销。一般C/S结构应用在广域网上。

随着Internet技术的兴起，又出现了一种对C/S结构改进的结构B/S（Browser/Server）结构，即浏览器和服务器结构，如图28.2所示。

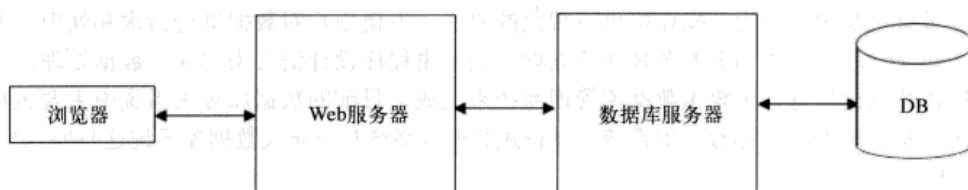


图28.2 B/S结构

在B/S结构中，通过WWW浏览器就可以实现用户的工作界面，将主要的业务逻辑处理放到了服务器端实现，从而减轻了客户端电脑载荷，降低了用户的成本。一般B/S结构应用在局域网上。

在B/S结构中，可以看到要处理的业务逻辑由客户端转到了服务器端，这就使得客户端变得越来越“瘦”，服务器端变得越来越“胖”，这种结构也有利于系统的升级和维护，降低系统维护的开销。

## 28.1.3 并行式数据库系统

并行式数据库系统（Parallel DBS）可以使用多个CPU和多个磁盘进行并行操作，从而提高数据处理和I/O速度。在有些应用中，数据库中数据量非常庞大，同时还要要求在极短的时间内处理大量的事务，这样的环境下，集中式数据库系统、客户端/服务器端数据库系统是无法胜任的。为了解决这样的问题，就出现了并行式数据库系统。

并行式数据库系统一般用于要求数据量显著提高，事务处理速度加快的场合。在并行式数据库系

统中，不是采用分时的方法执行操作，而是许多操作同时进行。并行式数据库系统中的两个重要性能指标是吞吐量和响应时间。所谓吞吐量，就是指给定时间间隔内可以完成多少个任务，所谓响应时间，是指完成一个任务需要消耗的时间。

#### 28.1.4 分布式数据库系统

分布式数据库系统（Distributed DBS）中数据的存储分散在不同的地方，但是这些数据在逻辑上仍然是一个整体，如同一个集中式数据库。在分布式数据库系统，存储在各个不同地方的数据库系统可以通过网络通信连接在一起。

在分布式数据库系统中，数据经网络的传输时间是影响查询的主要因素，数据传输量是衡量查询时间的一个主要指标。数据在网络中传输时，可以考虑将不参与操作的值或者无用的值不通过网络进行传输，通过这种方式可以优化分布式数据库系统中的数据查询。

### 28.2 数据库连接访问

在实际应用中，一般使用高级程序设计语言来完成业务逻辑的处理，使用数据库管理系统来完成数据的处理操作。高级程序设计语言要想访问数据库就必须与数据库进行连接。常用的连接访问方式包括ODBC、OLE DB、ADO、JDBC等。这一节就来介绍这几种常用的数据库连接访问方式。

#### 28.2.1 ODBC

ODBC（Open Database Connectivity）开放式数据库连接是微软公司开放服务结构（WOSA，Windows Open Services Architecture）中有关数据库的一个组成部分，它提供了一组对数据库访问的标准API（应用程序编程接口）。可以通过ODBC驱动程序访问多种数据源，与数据库进行连接。ODBC驱动程序是一些DLL，在这些DLL中提供了ODBC和数据库之间的接口。数据源是一种对数据连接的抽象，在数据源中包含了数据库位置和数据库类型等相关信息。ODBC本身也提供了对SQL语言的支持，用户可以直接将SQL语句送给ODBC。

ODBC的体系结构共分为四层：应用程序、驱动程序管理器、驱动程序和数据源。在应用程序层可以请求和终止与数据源的连接、向数据源发送SQL请求、处理错误、事务控制等操作。驱动程序管理器在Windows中是一个动态链接库，即ODBC.DLL，主要用于装入驱动程序。驱动程序层由微软、DBMS厂商或者第三方开发商提供，它必须符合ODBC的规范。例如，对于Oracle数据库，驱动程序为ORA6WIN.DLL；对于SQL Server数据库，驱动程序为SQLSRVR.DLL。驱动程序是实现ODBC函数和数据源之间交互的DLL，它可以对来自应用程序的ODBC函数调用进行响应。

#### 28.2.2 OLE DB

OLE DB（Object Link and Embed DB）对象连接和嵌入的是一个基于COM的数据存储对象，是一种开放式的标准，它规定了数据使用者和提供者之间的一种应用层协议。OLE DB中的对象主要包括数据源对象、阶段对象、命令对象和行组对象。

使用OLE DB的应用程序首先需要初始化OLE连接到数据源，然后发出命令并对结果进行处理，最后释放数据源对象并停止初始化OLE。

OLE DB可以提供对所有类型的数据的操作。主要是由数据提供者（Data Providers）、数据使用者



(Data Consumers) 和服务组件 (Service Components) 三个部分组成。数据提供者是指通过OLE DB提供数据，例如，SQL Server 数据库中的数据表就是数据提供者；数据使用者是指使用ADO 的应用程序或网页的程序或组件；服务组件用来执行数据提供者以及数据使用者之间数据传递的工作。

### 28.2.3 ADO

ADO (ActiveX Data Objects) ActiveX数据对象是一个用于存取数据源的COM组件，用以实现访问关系或非关系数据库中的数据。ADO为编程语言和统一数据访问方式OLE DB提供了一个中间层。开发人员在访问数据库时只需要关心到数据库的连接，而不需要关系数据库是如何实现的。

在.NET Framework中，微软也提供了一个面向Internet的版本的ADO，称为ADO.NET。ADO.NET提供了平台互用性和可伸缩的数据访问，支持对数据的松耦合访问，增强了对非连接编程模式的支持。ADO.NET中主要包含的对象包括SqlConnection 对象、Command对象、sqlDataReader对象、DataSet对象、SqlDataAdapter对象等。

- ☐ SqlConnection 对象：用于数据库的连接，连接时需要指定数据库服务器、数据库名字、用户名、密码等信息。
- ☐ Command对象：执行SQL语句并将发送SQL语句到数据库。在Command对象中包含了数据库系统的访问信息。
- ☐ sqlDataReader对象：取得从Command对象的SELECT语句中返回的结果，实现对数据的读取操作。
- ☐ DataSet对象：表示数据在内存中的表示形式。DataSet可以认为是一个数据容器。在DataSet中可以包括多个DataTable对象。
- ☐ SqlDataAdapter对象：用于从数据库中获取数据，并将其存储在DataSet中。它也可以取得DataSet中的更新数据，并将这些数据提交给数据库。

### 28.2.4 JDBC

JDBC (Java Database Connectivity) 是Java的数据库连接接口。它是JavaAPI中的一部分，JDBC允许任何使用Java语言编写的小应用程序或应用程序访问数据库，通过它可以将Java程序和关系数据库集成在一起。

JDBC与数据库的连接通信方式可以有两种形式，一种是利用JDBC-ODBC桥通过ODBC桥实现对数据库的访问，一种是通过数据库供应商提供的JDBC驱动程序实现对数据库的访问。在后一种访问中，Java应用程序首先是通过JDBC API与驱动管理器进行通信，再由驱动管理器中的驱动程序完成与数据库的通信。

JDBC API中主要的类和接口包括DriverManager类、Driver接口、Connection接口、Statement接口、DatabaseMetaData接口和ResultSet接口等，这些类和接口都在java.sql包中。

- ☐ DriverManager类：用来管理JDBC驱动程序，主要用于跟踪和加载驱动程序并负责选取数据库驱动程序和建立新的数据库连接。
- ☐ Driver接口：将API的调用映射到数据库。每个驱动程序类都必须实现该接口。
- ☐ Connection接口：用来将应用程序与指定的数据库连接在一起。
- ☐ Statement接口：用来执行静态SQL语句，并返回执行后的结果。
- ☐ DatabaseMetaData接口：返回与数据库和驱动程序等底层数据库相关的信息。

❑ **ResultSet接口**：提供对数据库表的访问，执行查询后返回的结果集。

JDBC驱动器类型可以有以下4种：JDBC-ODBC桥、本地API驱动、网络纯Java类库驱动和本地Java类库驱动。

❑ **JDBC-ODBC桥**：它是将JDBC转化为ODBC驱动，再利用ODBC对JDBC进行访问。通过使用ODBC可以实现编程语言与数据库进行交互的功能，这种驱动不是很常用。

❑ **本地API驱动**：可以将用户的调用转化为对数据库客户端相应API的调用。使用这种驱动需要在本地安装一些相关的代码。

❑ **网络纯Java类库驱动**：驱动程序与数据库服务器相互独立，通过向一个与数据库无关的协议将数据库请求发送给中间的某个服务器，再由该服务器实现对数据库的访问。与具体的数据库无关。

❑ **本地Java类库驱动**：将用户的请求转换为对数据库的协议请求，和数据库服务器直接进行通信。与具体的数据库无关。

## 28.3 Java与MySQL 5.0数据库连接与访问

Java作为目前的主流面向对象的程序设计语言之一，以其面向对象、跨平台、支持多线程和分布式等特点在Web应用程序开发、网络编程、手机游戏等各个方面都得到了广泛的应用。这一节就来介绍如何使用Java语言通过JDBC来实现对MySQL 5.0数据库的访问。

### 28.3.1 Java开发环境

在实现Java与MySQL 5.0数据库连接与访问之前，首先要安装Java开发环境。JDK（Java Development Kit）是一切Java应用程序的基础，所有的Java应用程序都是构建在这个基础之上的。它是由Sun公司提供的免费的软件包，该包中包括了Java程序的编写和运行所需要的工具。

可以到Sun公司的网站<http://Java.sun.com/>下载最新版本的JDK。本书使用的是JDK1.6.0\_10版本。从网上下载到该版本后即可安装了。具体安装过程如下：

（1）双击下载的jdk-6u10-rc-bin-b28-windows-i586-p-21\_jul\_2008安装文件，在弹出的“许可证协议”对话框中单击“接受”按钮，如图28.3所示。

（2）在“自定义安装”对话框中，会显示JDK的安装路径。这里使用默认的安装路径。如果想修改JDK的安装路径，可以单击对话框中的“更改”按钮，在弹出的“更改当前文件夹”对话框中对JDK的安装路径进行修改。如图28.4所示。

（3）在“自定义安装”对话框中单击“下一步”按钮后，程序会自定义安装JDK，安装JDK之后在出现的“目标文件夹”对话框中选择JVM的安装路径，这里使用默认的安装路径。如果想修改JDK的安装路径，可以单击对话框中的“更改”按钮，如图28.5所示。

（4）在“目标文件夹”对话框中单击“下一步”按钮，会出现“Java安装进度”对话框，显示安装

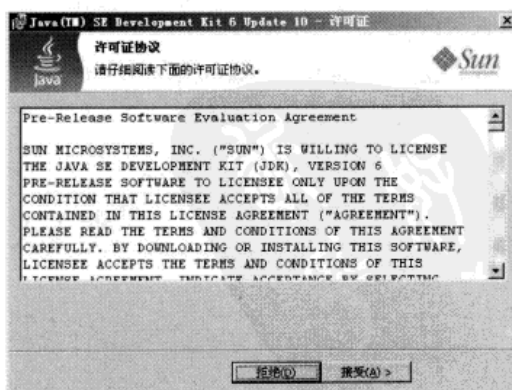


图28.3 “许可证协议”对话框

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

进度。安装完成后，会出现“完成”对话框，单击“完成”按钮即可完成JDK的安装，如图28.6所示。

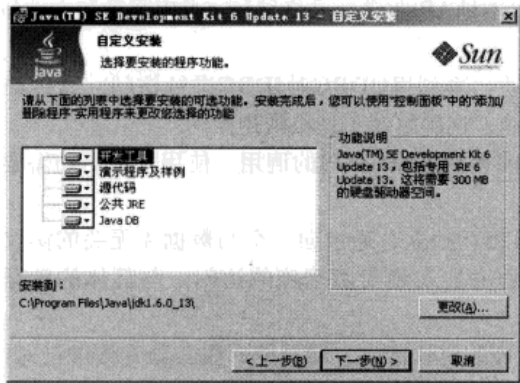


图28.4 “自定义安装”对话框

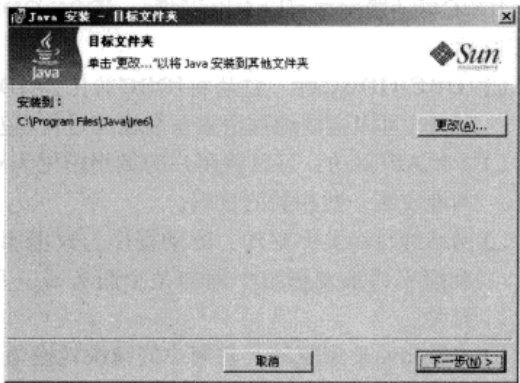


图28.5 “目标文件夹”对话框

(5) 在安装完成后还需要手动设置JDK环境变量。右击“我的电脑”图标，在弹出的快捷菜单中选择“属性”命令，在属性对话框中选择“高级”选项卡，如图28.7所示。

(6) 在“高级”对话框中单击“环境变量”按钮，在对话框下面的“系统变量”中找到变量名为Path的变量，单击“编辑”按钮，将JDK的安装路径“C:\Java\jdk1.6.0\_10\bin;”写入到变量名为path对应的变量值的最前面（注意在路径之后需要加一个分号），如图28.8所示。

(7) 设置完成后单击“确定”按钮，完成了环境变量的设置。

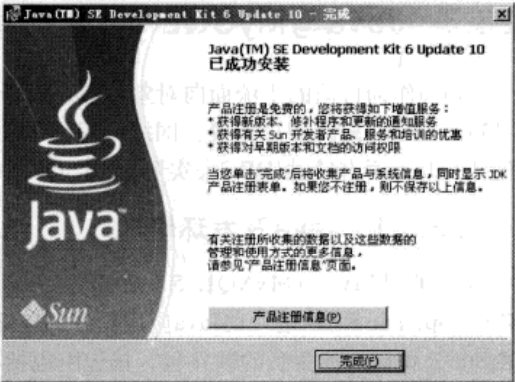


图28.6 “完成”对话框

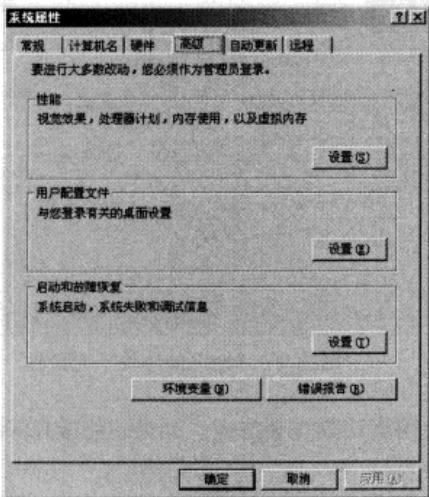


图28.7 系统属性-高级对话框

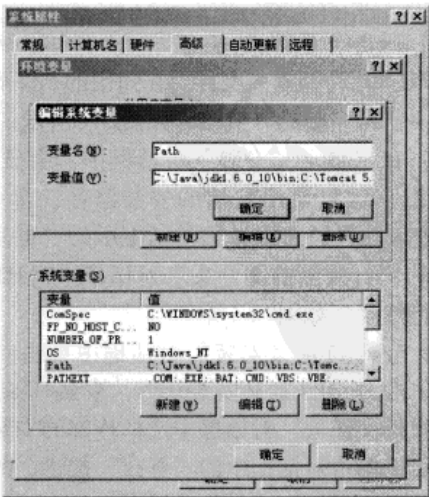


图28.8 系统变量的设置

28.3.2 安装Eclipse

Eclipse是一个开源的基于Java的免费开发平台，是一个可视化的开发环境。可以到其官方网站<http://download.eclipse.org/eclipse/downloads/>上下载Eclipse。Eclipse的安装非常简单，只要将下载的Eclipse压缩包解压缩到指定路径下，直接双击其中的eclipse.exe文件就可以运行Eclipse。其启动画面如图28.9所示。

第一次启动Eclipse会弹出一个对话框要求选择一个Eclipse的工作空间。这里选择的是“C:\eclipse\workspace”，如图28.10所示。

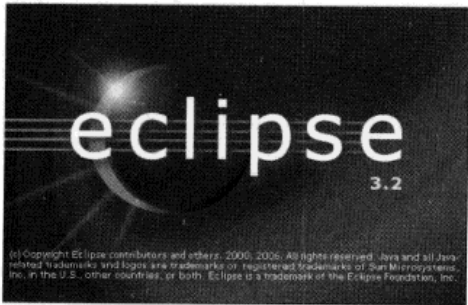


图28.9 Eclipse启动画面

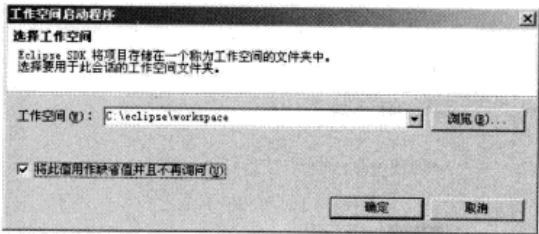


图28.10 选择Eclipse的工作空间

提示

如果想将该工作空间的路径作为其默认路径，可以把“将此值用作缺省值并且不再询问”前面的复选框按钮选中。这样，Eclipse在以后启动时就会将该路径作为默认路径。

单击“确定”按钮后，会出现一个“欢迎使用Eclipse3.2”的界面，此时表示Eclipse已经成功安装了。将其关闭后，就会出现Eclipse的工作区，如图28.11所示。

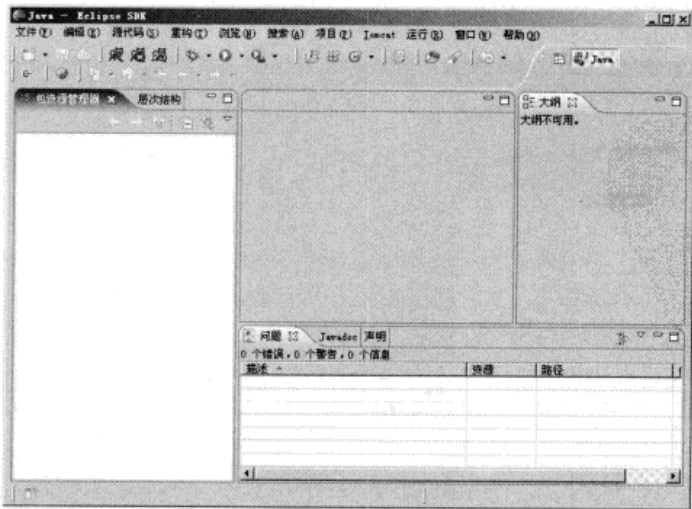


图28.11 Eclipse的工作区

在图28.11中好像没有看见编写代码的区域，别着急，如果想编辑代码，首先需要创建一个Java项

零基础学SQL

目。在Eclipse的工作区左侧的“包资源管理器”选项卡下的空白处单击鼠标右键，在弹出的快捷菜单中选择“新建”命令（或者选择“文件”|“新建”|“项目”命令），在出现的Java目录下选择Java项目，出现如图28.12所示的画面。

将“新建项目”中的“Java”文件夹展开，选中“Java项目”选项，单击“下一步”按钮。在出现的“新建Java项目”对话框中填写项目名称，这里设置的项目名称为DatabaseCon，如图28.13所示。

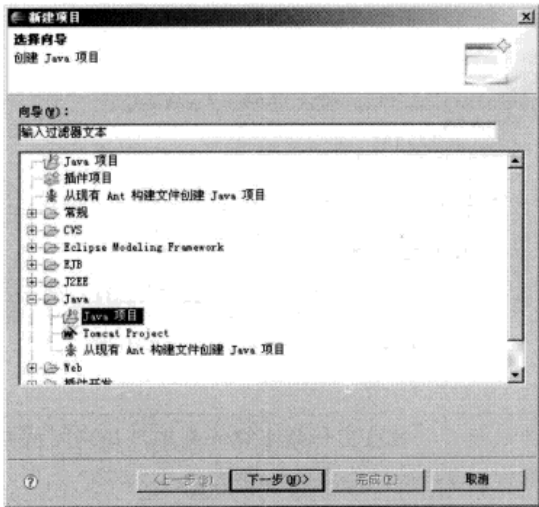


图28.12 选择Java项目

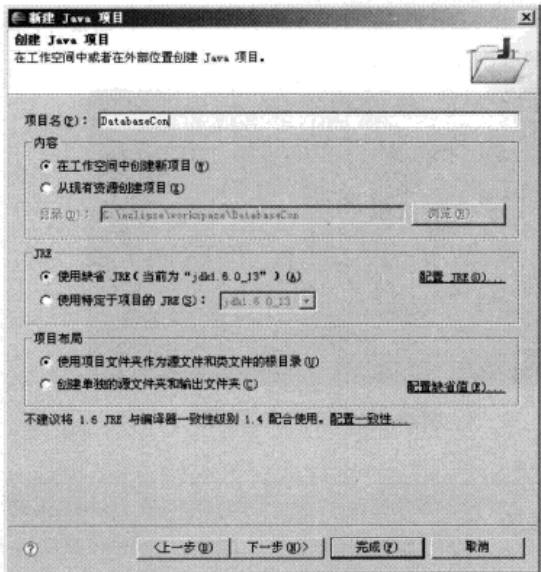


图28.13 填写项目名称

然后单击“完成”按钮。此时在Eclipse工作区的左侧就会多出一个项目名为DatabaseCon的目录，如图28.14所示。

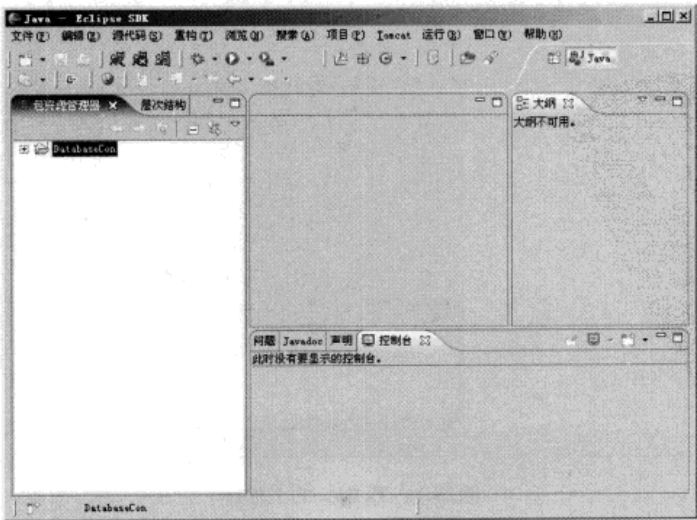


图28.14 DatabaseCon项目创建完成



28.3.3 添加MySQL 5.0驱动程序

为了实现Java与MySQL 5.0数据库的连接，还需要将MySQL 5.0的JDBC的驱动程序放到classpath指向的目录下。该驱动程序可以到MySQL的官方网站上<http://www.mysql.com/>下载。这里使用的JDBC的驱动程序为mysql-connector-java-5.0.5-bin.jar。在Eclipse中，可以使用以下步骤导入驱动程序。

- (1) 右击一个项目工程，在弹出的快捷菜单中选择“构建路径”|“配置构建路径选项”命令。
- (2) 在Java的构建路径对话框中，选择“库”选项，然后单击对话框右侧的“添加外部JAR”按钮。
- (3) 在出现的“选择JAR”对话框中，找到mysql-connector-java-5.0.5-bin.jar驱动程序所在的路径，将其选中，单击“打开”按钮，如图28.15所示。
- (4) 单击“确定”按钮，完成MySQL 5.0数据库驱动程序的添加。

在Eclipse下添加完该MySQL 5.0数据库驱动程序后，就可以对MySQL 5.0数据库进行连接以及数据的存取操作了。

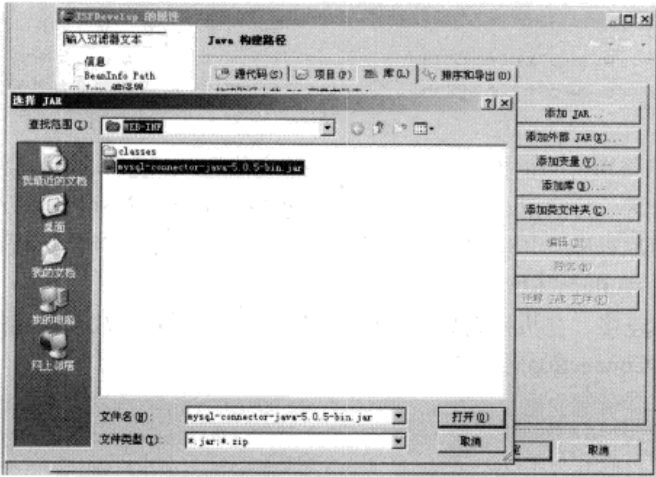


图28.15 添加MySQL 5.0驱动程序

28.3.4 Java与数据库的连接过程

在完成了Java开发环境的配置和MySQL 5.0数据库驱动程序的添加之后，就可以通过Java语言实现与MySQL 5.0数据库的连接了。Java与数据库的连接步骤如下：

- (1) 使用Class类加载当前数据源的驱动程序。

```
Class.forName(driver);
```

其中，driver表示数据库的驱动程序。MySQL 5.0数据库驱动程序为com.mysql.jdbc.Driver

- (2) 通过DriverManager类中的getConnection方法建立数据库的连接。

```
conn = DriverManager.getConnection(url, userName, password);
```

其中，url表示数据库的URL，MySQL数据库的URL为jdbc:mysql://[HOST]:[PORT]/[database\_name]。HOST表示本机，PORT表示端口号，database\_name表示数据库的名字。userName表示数据库使用的用



户名，password表示使用该数据库的密码。

(3) 使用Connection接口中的createStatement()方法创建Statement对象。

```
stmt = conn.createStatement();
```

其中，stmt表示一个Statement对象，通过该对象可以执行相应的SQL语句并将其发送给数据库。

(4) 执行SQL语句并将结果返回。执行的查询结果可以以ResultSet结果集的形式返回。

```
ResultSet rs = stmt.executeQuery(sqlSelect);           //执行SQL查询语句  
Int count = stmt.executeUpdate(sqlUpdate);             //执行SQL更新语句
```

其中，sqlSelect是一个用于执行查询的SQL语句，executeQuery()方法是用于执行查询SQL语句的方法，该方法返回一个结果集。sqlUpdate是一个用于执行更新的SQL语句（INSERT、UPDATE和DELETE），executeUpdate()方法是用于执行更新SQL语句的方法，该方法返回一个整数，用来表示数据表中数据更新的个数。

(5) 使用一个while循环获取ResultSet结果集中的所有记录。

```
while (rs.next()) {                                     //获取查询结果  
    String str1= rs.getString(1);                       //获取结果集中第一个对应字段的值  
    String str2= rs.getString(2);                       //获取结果集中第二个对应字段的值  
}
```

其中，rs.next()表示将指针的位置向下移动一行。该方法返回一个布尔值。当第一次调用该方法时，指针会指向该结果集的第一行的记录的位置。getString(1)表示取得ResultSet结果集中第一个对应字段的值，getString(2)表示取得ResultSet结果集中第二个对应字段的值。当指针指向的下一行记录不存在时，rs.next()返回false。while循环结束。

(6) 关闭数据库的连接。在执行完数据库的操作之后，需要将与数据库连接有关的对象关闭。主要包括Statement对象、Connection对象等。

```
stmt.close();  
conn.close();
```

## 28.4 Java与MySQL 5.0数据库开发

在实际应用中，需要通过一种程序设计语言实现对数据的存取操作。在28.3节中已经了解了Java开发环境以及Java与数据库的连接过程，本节通过实例介绍使用Java程序数据语言实现对MySQL 5.0数据库中数据的查询和更新操作。

### 28.4.1 查询数据

在使用Java与MySQL 5.0数据库开发之前，先看一个使用SQL语句执行查询操作的例子。例如，现在查询所有选修了t105这门课的学生的课程成绩。查询的SQL语句如下：

```
SELECT R.stuID,S.stuName,C.curID, C.curName,R.result  
FROM T_result R,T_curriculum C,T_student S  
WHERE R.curID=C.curID  
AND R.stuID=S.stuID  
AND C.curID = 't105'  
ORDER BY R.result
```

这个SQL语句要连接三个数据表，分别是课程成绩表T\_result、课程信息表T\_curriculum和学生信息表T\_student，并按照成绩从低到高进行排序，其在MySQL数据库中显示的结果如图28.16所示。

stuID	stuName	curID	curName	result
s253263	李风	t105	计算机系统结构	50
s206363	张明	t105	计算机系统结构	80
s102203	赵亮	t105	计算机系统结构	85

图28.16 查询所有选修了t105这门课的学生们的课程成绩

下面来介绍如何应用Java程序设计语言，通过与MySQL 5.0数据库连接实现上述的查询操作。在Java中，查询数据主要需要用到executeQuery()方法和ResultSet对象。executeQuery()方法用于执行给定的执行查询的SQL语句，该方法需要接收一个参数，这个参数是一个SELECT语句的字符串，其返回值是一个ResultSet对象。ResultSet对象是表示数据库结果集的数据表。下面来看一下使用Java程序数据语言如何实现对MySQL 5.0数据库中上述查询操作的方法，其程序代码如下所示。

```
import java.sql.*;
public class executeQueryTest {
    public static void main(String[] args) {
        String driver="com.mysql.jdbc.Driver";
        String url="jdbc:mysql://localhost:3306/ test_STInfo";
        String userName="root";
        String password=" root ";
        Connection conn = null;
        Statement stmt = null;
        try {
            Class.forName(driver);
        } catch(ClassNotFoundException e) {
            System.err.print("ClassNotFoundException");
        }
        try {
            conn = DriverManager.getConnection(url, userName, password);//建立数据库连接
            System.out.println("数据库连接成功");
            stmt = conn.createStatement();
            /*执行SQL查询语句*/
            String sqlSelect = " SELECT R.stuID,S.stuName,C.curID, C.curName,R.result"+
                                " FROM T_result R,T_curriculum C,T_student S "+
                                " WHERE R.curID=C.curID "+
                                " AND R.stuID=S.stuID "+
                                " AND R.stuID=S.stuID "+
                                " AND C.curID = 't105' "+
                                " ORDER BY R.result ";
            System.out.println("执行SQL查询语句为："+sqlSelect);
            ResultSet rs = stmt.executeQuery(sqlSelect);
            while (rs.next()) {
                String stuID = rs.getString(1);
                String stuName = rs.getString(2);
                String curID = rs.getString(3);
                String curName = rs.getString(4);
                String result = rs.getString(5);
                System.out.println(stuID + " , " + stuName + " , " + curID + " , " +
                                   curName + "," + result );
            }
        }
    }
}
```

```
    }
    } catch(SQLException e) {
        System.out.println("SQLException 异常"+e.getMessage());
        e.printStackTrace(); //追踪异常事件发生时执行堆栈内容
    } finally {
        try {
            stmt.close(); //关闭Statement
            if (conn != null)
                conn.close(); //关闭数据库连接
        } catch(SQLException e) {
            System.out.println("关闭数据库连接异常 "+e.getMessage());
            e.printStackTrace(); //追踪异常事件发生时执行堆栈内容
        }
    }
}
```

在这段Java代码中，首先加载驱动程序，建立数据库连接。然后创建Statement对象并执行SQL语句，这里的SQL语句是查询所有选修了t105这门课的学生们的课程成绩。使用Statement中的executeQuery()方法执行SQL查询操作并将结果返回到一个ResultSet结果集。

在while循环中通过使用next方法将ResultSet对象中的数据逐一打印输出。如果在ResultSet对象中没有下一行，则会返回false，并终止while循环。在finally语句块中关闭Statement语句和数据库的连接。在代码中的try...catch...finally是Java程序的异常处理。其运行结果在Eclipse的控制台中显示，如图28.17所示。

从这个控制台中显示的内容可以看到，其显示的结果与图28.16中使用SQL语句在MySQL 5.0数据库中查询的结果是完全相同的。

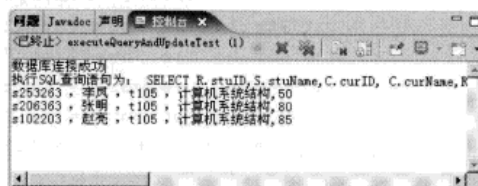


图28.17 查询所有选修了t105这门课的学生们的课程成绩

## 28.4.2 更新数据

在Java中，更新数据库中的数据需要用到executeUpdate()方法。executeUpdate()方法用于执行给定DML语句（例如，INSERT、UPDATE和DELETE）并返回一个int类型的整数。该方法需要接收一个参数，这个参数是一个DML语句的字符串，返回的整数表示插入数据的数量。例如，现在要修改课程成绩表T\_result中的数据，将学生编号为s253263的学生t105这门课的成绩修改为60分。可以通过以下的代码来实现。

```
import java.sql.*;
public class executeUpdateTest {
    public static void main(String[] args) {
        String driver="com.mysql.jdbc.Driver";
        String url="jdbc:mysql://localhost:3306/ test_STInfo";
        String userName="root"; //数据库用户名
        String password=" root "; //数据库密码
        Connection conn = null; //数据库连接对象
    }
}
```



```
Statement stmt = null;                                //执行SQL对象
try {
    Class.forName(driver);                            //加载驱动器
} catch(ClassNotFoundException e) {
    System.err.print("ClassNotFoundException");
}
try {
    conn = DriverManager.getConnection(url, userName, password); //建立数据库连接
    System.out.println("数据库连接成功");
    stmt = conn.createStatement();                    //创建Statement对象
    /*执行SQL查询语句*/
    String sqlUpdate= " UPDATE T_result SET result = 60 "+
        " WHERE stuID = 's253263' "+
        " AND curID = 't105' " ;
    System.out.println("执行SQL更新语句为："+ sqlUpdate); //打印输出SQL更新语句
    int count = stmt.executeUpdate (sqlUpdate);           //执行SQL查询语句
    System.out.println("修改操作执行完成");
    System.out.println("使用修改操作执行的数据行为 "+count);
    /*执行SQL查询语句*/
    String sqlSelect = " SELECT stuID,curID, result"+
        " FROM t_result t "+
        " WHERE stuID = 's253263' "+
        " AND curID = 't105' " ;
    System.out.println("执行SQL查询语句为："+sqlSelect); //打印输出SQL查询语句
    ResultSet rs = stmt.executeQuery(sqlSelect);          //执行SQL查询语句
    while (rs.next()) {                                    //查询结果
        String stuID = rs.getString(1);                  //学生编号
        String curID = rs.getString(2);                  //课程编号
        String result = rs.getString(3);                 //课程成绩
        System.out.println(stuID + " , " + curID + " , " +result );
    }
} catch(SQLException e) {
    System.out.println("SQLException 异常"+e. getMessage());
    e.printStackTrace();                                //追踪异常事件发生时执行堆栈内容
} finally {
    try {
        stmt.close();                                    //关闭Statement
        if (conn != null)
            conn.close();                                //关闭数据库连接
    } catch(SQLException e) {
        System.out.println("关闭数据库连接异常 "+e. getMessage());
        e.printStackTrace();                            //追踪异常事件发生时执行堆栈内容
    }
}
}
```

这段代码与28.4.1小节中的代码基本相同，在更新数据的操作中，字符串sqlUpdate是一个用来修改T\_result表中指定学生编号和课程编号的课程成绩的UPDATE语句。使用Statement中的executeUpdate()



零基础学SQL

方法执行SQL更新操作，该方法返回一个整数，用于对更新操作中执行的行数进行统计。在更新数据完成之后，使用SELECT语句对T\_result表重新进行查询，并将查询的结果显示到Eclipse的控制台中。其显示结果如图28.18所示。

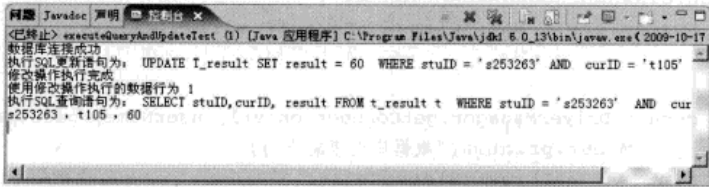


图28.18 学生编号为s253263的学生t105这门课的成绩修改为60分

从图28.18中可以看到，执行完executeUpdate()方法之后，学生编号为s253263的学生t105这门课的课程成绩数据已经在数据库中被修改了。使用executeUpdate()方法同样也可以完成INSERT语句和DELETE语句，其执行完成与修改数据的执行完成一样，也是通过使用executeUpdate()方法来完成。只需要将executeUpdate()方法接收的参数修改为INSERT语句和DELETE语句的字符串就可以实现INSERT和DELETE操作了。读者可以自己动手试一下，这里就不再举例。

**说明** 这里介绍的只是与数据库连接和访问有关的一些方法，并没有对Java语言中的代码做过多详细的讲解。有关Java程序设计语言的具体内容可以参考有关讲解Java的书籍。

28.5 小结

本章主要介绍了有关实现数据库连接和访问的内容，并通过一种具体的Java语言和一个具体的数据库讲解了数据库连接以及如何对数据库中的数据进行存取操作。在Java程序设计语言中的JDBC的功能还有很多，如果想进一步了解JDBC，可以参考有关Java的数据库编程的书籍。

当然，在实际应用中，不仅可以使⽤Java与MySQL数据库，也可以使用其他的程序设计语言（例如，ASP、VB、C#、Delphi等）和不同的数据库（例如，Oracle、Microsoft SQL Server、PostgreSQL等）进行连接和数据存取访问。由于篇幅所限，这里不可能对每一种语言和每一个数据库的连接访问都进行介绍。



## 附录A 常用SQL语句

附录A中介绍的是常用的SQL语句，由于一些SQL语句在不同的数据库的实现会有所不同，需要查询相关的文档。

### A.1 数据定义语言（DDL）

#### A.1.1 创建数据库（CREATE DATABASE）

```
CREATE DATABASE database_name
```

#### A.1.2 创建数据表（CREATE TABLE）

##### 1. 创建数据库中的表

```
CREATE TABLE table_name(  
column_name1 datatype1 [constraint_condition1]  
[, column_name2 datatype2 [constraint_condition2]]...  
)
```

##### 2. 基于一个数据表创建另一个数据表

```
CREATE TABLE table_name  
AS  
SELECT column_name1, column_name2,...  
FROM table_name1, table_name2...  
[WHERE conditions  
GROUP BY column_name1, column_name2,...  
HAVING conditions  
ORDER BY column_name1, column_name2,...  
]
```

#### A.1.3 创建索引（CREATE INDEX）

```
CREATE [UNIQUE][[CLUSTER]INDEX index_name  
ON table_name (column_name [ASC|DESC])
```

#### A.1.4 创建视图（CREATE VIEW）

```
CREATE VIEW view_name([column_name1[,column_name2]...]  
AS
```

## 零基础学SQL

```
SELECT column_name1, column_name2,...
FROM table_name1, table_name2...
[WHERE conditions
GROUP BY column_name1, column_name2,...
HAVING conditions
ORDER BY column_name1, column_name2,...
]
```

### A.1.5 修改基本表 (ALTER TABLE)

```
ALTER TABLE table_name
[ADD(column_name datatype [constraint_condition])]
[MODIFY column_name datatype]
[DROP constraint_type]
```

### A.1.6 删除数据库 (DROP DATABASE)

```
DROP TABLE table_name[CASCADE CONSTRAINTS]
```

### A.1.7 删除数据表 (DROP TABLE)

```
DROP DATABASE database_name
```

### A.1.8 删除索引 (DROP INDEX)

```
DROP INDEX index_name
```

### A.1.9 删除视图 (DROP VIEW)

```
DROP VIEW view_name
```

## A.2 数据查询语言 (DQL)

### SELECT 语句

```
SELECT column_name1, column_name2,...
FROM table_name1, table_name2...
[WHERE conditions
GROUP BY column_name1, column_name2,...
HAVING conditions
ORDER BY column_name1, column_name2,...
]
```

## A.3 数据操作语言（DML）

### A.3.1 插入数据（INSERT）

#### 1. 插入单行记录

```
INSERT INTO table_name[(column_name1,column_name2...)]  
VALUES (value1[,value2]...)
```

#### 2. 基于一个表向另一个数据表中插入数据

```
INSERT INTO table_name[(column_name1,column_name2...)]  
SELECT column_name1, column_name2,...  
FROM table_name1, table_name2...  
[WHERE conditions]
```

### A.3.2 修改数据(UPDATE)

```
UPDATE table_name  
SET column 1= value1[,column 2=v alue2]  
WHERE condition
```

### A.3.3 删除数据(DELETE)

```
DELETE FROM table_name  
[WHERE condition]
```

## A.4 数据控制语言（DCL）

### A.4.1 分配权限（GRANT）

```
GRANT 权限[,权限] ON TABLE 表名[,表名]  
TO 用户[,用户]  
[WITH GRANT OPTION]
```

### A.4.2 回收权限（REVOKE）

```
REVOKE权限[,权限] ON TABLE 表名[,表名]  
FROM 用户[,用户]
```

## A.5 事务控制语言

### A.5.1 创建事务保存点（SAVEPOINT）

#### 1. Oracle数据库

```
SAVEPOINT保存点
```

## 2. Microsoft SQL Server数据库

SAVE TRANSACTION 保存点

### A.5.2 提交事务 (COMMIT)

COMMIT [事务名]

### A.5.3 回滚事务 (ROLLBACK)

ROLLBACK [TO 保存点]

## A.6 创建存储过程

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(argument1[IN|OUT|IN OUT] datatype1 [,argument2[IN|OUT|IN OUT] datatype2...])
{IS | AS}
[local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [procedure_name];
```

## A.7 创建函数

```
CREATE [OR REPLACE] FUNCTION function_name
[(argument1[IN|OUT|IN OUT] datatype1 [,argument2[IN|OUT|IN OUT] datatype2...])
RETURN return_datatype
{IS | AS}
[local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [function_name];
```

## A.8 创建包

### A.8.1 创建包说明

```
CREATE [OR REPLACE] PACKAGE package_name
{IS | AS}
    [variable_declaration ...]      -- 声明变量
    [cursor_declaration...]         -- 声明游标
    [exception_declaration ...]     -- 声明异常
```



```
[object_declaration ...]      -- 声明对象
[function_declaration...]      -- 声明函数
[procedure_declaration...]     -- 声明过程
END [package_name];
```

## A.8.2 创建包体

```
CREATE [OR REPLACE] PACKAGE BODY package_name
{IS | AS}
    -- 公有变量、常量、游标、自定义数据类型、存储过程或者函数的实现
    -- 定义私有的变量、常量、游标、自定义数据类型、存储过程或者函数
END package_name;
```

## A.9 创建触发器

```
CREATE OR REPLACE TRIGGER trigger_name
{ BEFORE|AFTER| INSTEAD OF } trigger_event
ON {table_name | view_name|database | schema}
FOR EACH ROW
[WHEN (logical_expression)]
[DECLARE
    declaration_statements; ]
BEGIN
    execution_statements;
END [trigger_name];
```

## 附录B 常用函数对照

### B.1 聚合函数（统计函数）

B.1 聚合函数（统计函数）及其功能说明

函数功能	MySQL函数	SQL Server函数	Oracle函数
求平均值	AVG ()	AVG ()	AVG ()
计算行数	COUNT()	COUNT()	COUNT()
求最大值	MAX()	MAX()	MAX()
求最小值	MIN()	MIN()	MIN()
求总和	SUM()	SUM()	SUM()

### B.2 字符函数

B.2 字符函数及其功能说明

函数功能	MySQL函数	SQL Server函数	Oracle函数
取得字符的ASCII码	ASCII(string)	ASCII(string)	ASCII(string)
字符串连接	CONCAT(string1, string2,...)	string1+string2	CONCAT(string1, string2)
将ASCII码转换为相应字符		CHR(N)	CHR(N)
取得指定的子串在字符串中的位置	INSTR(string,substring)	CHARINDEX(sub-string, string,[start])	INSTR(string,substring, [,n [,m]])
将字符串全部转换为小写	LOWER(string)	LOWER(string)	LOWER(string)
将字符串全部转换为大写	UPPER(string)	UPPER(string)	UPPER(string)
去除字符串左侧空格	LTRIM(string,[set])	LTRIM(string,[set])	LTRIM(string,[set])
去除字符串尾空格	RTRIM(string,[set])	RTRIM(string,[set])	RTRIM(string,[set])
替换指定的子串	REPLACE(string, search_string [,replace_string])		REPLACE(string, search_string [,replace_string])
颠倒指定字符串的顺序	REVERSE(string)	REVERSE(string)	REVERSE(string)
匹配字符串的语音表示	SOUNDEX(string)	SOUNDEX(string)	SOUNDEX(string)
将字符串重复指定次数	REPEAT(string ,count)	REPLICATE(str,count)	
替换指定的子串	REPLACE(string, search_string [,replace_string])	STUFF(string , start, length , replace_string)	REPLACE(string, search_string [,replace_string])

(续)

函数功能	MySQL函数	SQL Server函数	Oracle函数
左侧填充空格或者字符	LPAD(string,length,padstring)		LPAD(string,length,padstring)
右侧填充空格或者字符	RPAD(string,length,padstring)		RPAD(string,length,padstring)
截取字符串	SUBSTRING(string FROM start[FOR length])	SUBSTRING(string FROM start[FOR length])	SUBSTR(string, start [,length])
从指定字符串左侧读取子串	LEFT(string,length)	LEFT(string,length)	
从指定字符串右侧读取子串	RIGHT(string,length)	RIGHT(string,length)	
将字符串中单词的首字母转换为大写			INITCAP(string)
计算字符串长度	LENGTH(string)	LEN(string)	LENGTH(string)
比较两个字符串	STRCMP(string1,string2)		

## B.3 数字函数

B.3 数字函数及其功能说明

函数功能	MySQL函数	SQL Server函数	Oracle函数
求绝对值	ABS(n)	ABS(n)	ABS(n)
求反余弦值	ACOS(n)	ACOS(n)	ACOS(n)
求反正弦值	ASIN(n)	ASIN(n)	ASIN(n)
求余弦值	COS(n)	COS(n)	COS(n)
求余切值	COT(n)	COT(n)	COT(n)
取得大于等于指定数的最小整数	CEIL(n)或者CEILING (n)	CEILING(n)	CEIL(n)
取得小于等于指定值的最大整数	FLOOR(n)	FLOOR(n)	FLOOR(n)
弧度转换为角度	DEGREES(n)	DEGREES(n)	
将角度转换为弧度	RADIANS(n)	RADIANS(n)	
以e为底的幂运算	EXP(n)	EXP(n)	EXP(n)
求自然对数	LOG(n)	LOG(n)	LN(n)
求以10为底对数	LOG10(n)	LOG10(n)	LOG(10,n)
求余数	MOD(n,m)	%	MOD(n,m)
求π值	PI()	PI()	
以其他任意数为底的幂运算	POWER(a,b)或者POW(a,b)	POWER(a,b)	POWER(a,b)
求平方	POWER (m,n)	SQUARE(n)	POWER (m,n)
求四舍五入值	ROUND(n [,m])	ROUND(n [,m])	ROUND(n [,m])
求正弦值	SIN(n)	SIN(n)	SIN(n)
取得指定值的符号标志	SIGN(n)	SIGN(n)	SIGN(n)
求平方根	SQRT(n)	SQRT(n)	SQRT(n)
求正切值	TAN(n)	TAN(n)	TAN(n)
对指定值进行截取操作	TRUNCATE(n[,m])		TRUNC(n[,m])
求集合中的最大值	GREATEST(n1,n2,n3...)		GREATEST(n1,n2,n3...)
求集合中的最小值	LEAST(n1,n2,n3...)		LEAST(n1,n2,n3...)



## B.4 日期函数

B.4 日期函数及其功能说明

函数功能	MySQL函数	SQL Server函数	Oracle函数
取得当前系统的日期和时间	NOW()或者SYSDATE()	GETDATE()	SYSDATE
对日期值进行加减运算	DATE_ADD(date,INTERVAL expression type)	DATEADD (date-part,number,date)	ADD_MONTH(date,n)
取得日期之后指定工作日对应的日期			NEXT_DAY(date,n)
取得日期值中年的整数	YEAR()	YEAR()	EXTRACT(date FROM datetime)
取得日期值中月的整数	MONTH ()	取得日期值中年的整数	EXTRACT(date FROM datetime)
取得日期值中日的整数	DAY()	DAY()	EXTRACT(date FROM datetime)
取得两日期时间间隔	DATEDIFF()	DATEDIFF()	MONTHS_BETWEEN(date1,date2)
取得月的最后一天	LAST_DAY(SYSTEM)		LAST_DAY(SYSTEM)
日期时间的舍入		CONVERT()	ROUND (date[,fmt])
截断指定的日期时间		CONVERT()	TRUNC(date[,fmt])
时区转换			NEW_TIME()
取得日期值中指定部分的整数		DATEPART()	TO_NUMBER(TO_CHAR())

## B.5 类型转换函数

B.5 类型转换函数及其功能说明

函数功能	MySQL函数	SQL Server函数	Oracle函数
字符转换函数	CAST(expression AS datatype[length])	STR (float[,length [,decima]]	TO_CHAR(n[,fmt[,nlsparams']])
日期转换函数	DATE_FORMAT(date,fmt)	CONVERT (datatype[length], date[,style])	TO_DATE(string[,fmt[,nlsparams']])
数值转换函数		CONVERT()	TO_NUMBER(char[,fmt[,nlsparams']])

## B.6 空值处理与分支函数

B.6 空值处理与分支转换函数及其功能说明

函数功能	MySQL函数	SQL Server函数	Oracle函数
处理NULL值	ISNULL(expression) 或者 COALESCE(expression1, expression2,expression3,...)	ISNULL(expression)	NVL(expression1,expression2) 或者 NVL2(expression1,expression2,expression3) 或者 COALESCE(expression1, expression2,expression3,...)
分支控制	IF(expression,result1,result2)		DECODE(expression,search1, result1[,search2,result2]...[default_result])
多分支控制	CASE表达式	CASE表达式	CASE表达式